Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

November 6, 2018

Contents

theory *Bits-Natural*
imports
Refine-Imperative-HOL.IICF
HOL−Word.Bits-Bit

HOL−Word.Bool-List-Representation
begin

instantiation *nat* :: *bits*
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*: ‹$m << n = m * (2^\frown n)$› **for** $m :: nat$
  ⟨*proof*⟩

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

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

instantiation *natural* :: *bits*
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 $(m >> n) = (\textit{integer-of-natural } m) >> n$
  ⟨*proof*⟩

lemma [*code*]:
  integer-of-natural $(m << n) = (\textit{integer-of-natural } m) << n$
  ⟨*proof*⟩

end

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

lemma *bitAND-1-mod-2*: ‹$\textit{bitAND } L\ 1 = L \textit{ mod } 2$› **for** $L :: \textit{nat}$
  ⟨*proof*⟩

lemma *shiftl-0-uint32*[*simp*]: ‹$n << 0 = n$› **for** $n :: \textit{uint32}$
  ⟨*proof*⟩

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

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

lemma *nat-test-bit0-iff*: ‹$n\ !!\ 0 \longleftrightarrow n \textit{ mod } 2 = 1$› **for** $n :: \textit{nat}$
⟨*proof*⟩
lemma *test-bit-2*: ‹$m > 0 \Longrightarrow (2{*}n)\ !!\ m \longleftrightarrow n\ !!\ (m - 1)$› **for** $n :: \textit{nat}$
  ⟨*proof*⟩

lemma *test-bit-Suc-2*: ‹$m > 0 \Longrightarrow \textit{Suc } (2 * n)\ !!\ m \longleftrightarrow (2 * n)\ !!\ m$› **for** $n :: \textit{nat}$
  ⟨*proof*⟩

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

lemma *bin-sc-ge0*: ‹$w >= 0 ==> (0{::}\textit{int}) \leq \textit{bin-sc } n\ b\ w$›
  ⟨*proof*⟩

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

lemma *nat-bin-nth-bl*: $n < m \Longrightarrow w\ !!\ n = \textit{nth } (\textit{rev } (\textit{bin-to-bl } m\ (\textit{int } w)))\ n$ **for** $w :: \textit{nat}$
  ⟨*proof*⟩

lemma *bin-nth-ge-size*: ‹$\textit{nat } na \leq n \Longrightarrow 0 \leq na \Longrightarrow \textit{bin-nth } na\ n = \textit{False}$›
⟨*proof*⟩

lemma *test-bit-nat-outside*: $n > \textit{size } w \Longrightarrow \neg w\ !!\ n$ **for** $w :: \textit{nat}$

⟨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
Refine-Imperative-HOL.IICF
Weidenbach-Book-Base.WB-List-More
begin

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⟩

no-notation *Ref.update* (- := - *62*)

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** *simp* =
((*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-*$\Downarrow$ = ‹
Scan.succeed (*Thm.rule-attribute* [] (*More-Refinement.down-converse o Context.proof-of*))
› *convert theorem from* @{*text* →}−*form to* @{*text* $\Downarrow$}−*form.*

method *to-*$\Downarrow$ =
(*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**
‹$V \leq \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge P\ T1 \wedge Q'\ T1\ T0\}\ U$› **and**
‹$W \leq \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$›
shows ‹$W \leq \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$›
⟨*proof*⟩

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

lemma *Down-add-assumption-beginning*:
assumes
‹*nofail U*› **and**
‹$V \leq \Downarrow \{(T1,\ T0).\ P\ T1 \wedge Q'\ T1\ T0\}\ U$› **and**
‹$W \leq \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$›
shows ‹$W \leq \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$›
⟨*proof*⟩

lemma *Down-add-assumption-beginning-single*:
assumes
‹*nofail U*› **and**
‹$V \leq \Downarrow \{(T1,\ T0).\ P\ T1\}\ U$› **and**
‹$W \leq \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$›
shows ‹$W \leq \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$›
⟨*proof*⟩

lemma *Down-del-assumption-beginning*:
fixes U :: ‹*'a nres*› **and** V :: ‹*'b nres*› **and** $Q\ Q'$ :: ‹$'b \Rightarrow 'a \Rightarrow bool$›
assumes
‹$V \leq \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge Q'\ T1\ T0\}\ U$›
shows ‹$V \leq \Downarrow \{(T1,\ T0).\ Q'\ T1\ T0\}\ U$›
⟨*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* ⇒ *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* ⇒ *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*]: ‹$S1 \le \Downarrow \{(T1,\ T0).\ Q\ T0\ T1 \wedge Q'\ T1\ T0\}\ S0$› (*multi*) **and**
J: - **for** *S1*:: ‹′*b nres*› **and** *S0* :: ‹′*a nres*› **and** *Q Q′* ⇒
‹*match J*[*uncurry*] *in*
J[*curry*]: ‹$- \Longrightarrow S2 \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ S1$› *for S2* :: ‹′*c nres*› *and R* ⇒
‹*insert Down-del-assumption-beginning*[*where* Q = ‹$\lambda\ x\ y.\ Q\ y\ x$› *and* $Q' = Q'$, *OF I*];
unify-Down-invs2-normalisation-post›
| - ⇒ ‹*fail*››
)

Example:

lemma
assumes
‹*nofail S0*› **and**
1: ‹$S1 \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge P\ T1 \wedge P'\ T1 \wedge P'''\ T1 \wedge Q'\ T1\ T0 \wedge P42\ T1\}\ S0$› **and**
2: ‹$S2 \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ S1$›
shows ‹$S2$
$\le \Downarrow \{(T2,\ T1).$
$R\ T2\ T1\ \wedge$
$P\ T1 \wedge P'\ T1 \wedge P'''\ T1 \wedge P42\ T1\}$
$S1$›
⟨*proof*⟩

Inversion Tactics

lemma *refinement-trans-long*:
‹$A = A' \Longrightarrow B = B' \Longrightarrow R \subseteq R' \Longrightarrow A \le \Downarrow R\ B \Longrightarrow A' \le \Downarrow R'\ B'$›
⟨*proof*⟩

lemma *mem-set-trans*:

$\langle A \subseteq B \Longrightarrow a \in A \Longrightarrow a \in B\rangle$
⟨*proof*⟩

lemma *fun-rel-syn-invert*:
$\langle a = a' \Longrightarrow b \subseteq b' \Longrightarrow a \rightarrow b \subseteq a' \rightarrow b'\rangle$
⟨*proof*⟩

lemma *fref-syn-invert*:
$\langle a = a' \Longrightarrow b \subseteq b' \Longrightarrow a \rightarrow_f b \subseteq a' \rightarrow_f b'\rangle$
⟨*proof*⟩

lemma *nres-rel-mono*:
$\langle a \subseteq a' \Longrightarrow \langle a\rangle$ *nres-rel* $\subseteq \langle a'\rangle$ *nres-rel*$\rangle$
⟨*proof*⟩

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

method *match-fun-rel* =
((*match* **conclusion in**
$\langle - \rightarrow - \subseteq - \rightarrow -\rangle \Rightarrow$ ⟨*rule fun-rel-mono*⟩
| $\langle - \rightarrow_f - \subseteq - \rightarrow_f -\rangle \Rightarrow$ ⟨*rule fref-syn-invert*⟩
| $\langle\langle - \rangle$*nres-rel* $\subseteq \langle - \rangle$*nres-rel*$\rangle \Rightarrow$ ⟨*rule nres-rel-mono*⟩
| $\langle [-]_f\ - \rightarrow - \subseteq [-]_f\ - \rightarrow -\rangle \Rightarrow$ ⟨*rule fref-mono*⟩
)+)

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

method *match-spec-trans* =
(*match* **conclusion in** $\langle f \leq SPEC\ R\rangle$ **for** f :: ⟨'*a nres*⟩ **and** R :: ⟨'*a* ⇒ *bool*⟩ ⇒
⟨*print-term f*; *match premises in I*: $\langle - \Longrightarrow - \Longrightarrow f' \leq SPEC\ R'\rangle$ *for* f' :: ⟨'*a nres*⟩ *and* R' :: ⟨'*a* ⇒ *bool*⟩
$\Rightarrow$ ⟨*print-term* f'; *rule weaken-SPEC2*[*of* $f'\ R'\ f\ R$]⟩⟩)

0.0.2 More Notations

abbreviation *comp4* (**infixl** *oooo 55*) **where** *f oooo g* $\equiv \lambda x.$ *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 oooo* (*g x*)
abbreviation *comp7* (**infixl** *ooooooo 55*) **where** *f ooooooo g* $\equiv \lambda x.$ *f oooo* (*g x*)
abbreviation *comp8* (**infixl** *oooooooo 55*) **where** *f oooooooo g* $\equiv \lambda x.$ *f oooo* (*g x*)

notation
comp4 (**infixl** ∘∘∘ *55*) **and**
comp5 (**infixl** ∘∘∘∘ *55*) **and**
comp6 (**infixl** ∘∘∘∘∘ *55*) **and**
comp7 (**infixl** ∘∘∘∘∘∘ *55*) **and**
comp8 (**infixl** ∘∘∘∘∘∘∘ *55*)

notation *prod-assn* (**infixr** $*a$ *90*)

0.0.3 More Theorems for Refinement

lemma *prod-assn-id-assn-destroy*: ‹$R^d *_a$ *id-assn*^{d} = ($R *a$ *id-assn*)^d›
⟨*proof*⟩

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

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

lemma *list-assn-map-list-assn*: ‹*list-assn* g (*map* $f\ x$) xi = *list-assn* $(\lambda a\ c.\ g\ (f\ a)\ c)\ x\ xi$›
⟨*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*: ‹$(\bigwedge x.\ x \in \Phi \Longrightarrow f\ x \le \Downarrow R\ M) \Longrightarrow M' \le RES\ \Phi \Longrightarrow M' \gg\!= f \le \Downarrow R\ M$›
⟨*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*:
‹$REC_T\ (WHILEI\text{-}body\ (\gg\!=)\ RETURN\ I'\ b'\ f)\ x' =$
$(REC_T\ (WHILEI\text{-}body\ (\gg\!=)\ RETURN\ (\lambda x'.\ I'\ x' \wedge (b'\ x' \longrightarrow f\ x' = FAIL \vee f\ x' \le SPEC\ I'))\ b'\ f)\ x')$›
(**is** ‹$REC_T\ ?f\ x' = REC_T\ ?f'\ x'$›)
⟨*proof*⟩

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

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

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

lemma *WHILEIT-rule-stronger-inv-RES*:
assumes
‹$wf\ R$› **and**
‹$I\ s$› **and**
‹$I'\ s$›
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \le SPEC\ (\lambda s'.\ I\ s' \wedge\ I'\ s' \wedge (s',\ s) \in R)$› **and**
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow \neg\ b\ s \Longrightarrow s \in \Phi$›
shows ‹$WHILE_T{}^I\ b\ f\ s \le RES\ \Phi$›
⟨*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*: ‹$(uncurry\ (return\ oo\ (\lambda a\ b.\ Pair\ a\ b)),\ uncurry\ (RETURN\ oo\ (\lambda a\ b.\ Pair\ a\ b))) \in$

$A^d *_a B^d \rightarrow_a$ *prod-assn A B*›
⟨*proof*⟩

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

lemma *bind-rule-complete-RES*: ‹$(M \gg= f \leq RES\ \Phi) = (M \leq SPEC\ (\lambda x.\ f\ x \leq RES\ \Phi))$›
⟨*proof*⟩

This version works only for *pure* refinement relations:

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

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

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

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

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

lemma *fref-to-Down-curry2'*:
‹$(uncurry2\ f,\ uncurry2\ g) \in A \rightarrow_f \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'.\ (((x,\ y),\ z),\ ((x',\ y'),\ z')) \in A \Longrightarrow$
$f\ x\ y\ z \leq \Downarrow B\ (g\ x'\ y'\ z'))$›
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

More Simplification Theorems

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

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

lemma *WHILEIT-refine-with-post*:
assumes *R0*: $I'\ x' \Longrightarrow (x,x') \in R$
assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x') \in R;\ I'\ x'\ ]\!] \Longrightarrow I\ x$

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

0.0.4 Some Refinement

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

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*:
‹$\{(x, y).\ P\ x\ y\}\ O\ \{(c, a).\ a = f\ c\} = \{(x, c).\ \exists y.\ P\ x\ y \land c = f\ y\}$ ›
⟨*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: ‹$A \models$ *list-mset-assn R N* (*ab* # *list*)›
shows ‹$\exists ab'.\ (ab, ab') \in$ *the-pure R* $\land ab' \in\# N \land A \models$ *list-mset-assn R* (*remove1-mset ab' N*) (*list*)›
⟨*proof*⟩

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

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

lemma *refine-add-invariants'*:
assumes
‹$f\ S \le \Downarrow \{(S, S').\ Q'\ S\ S' \land Q\ S\}\ gS$› **and**
‹$y \le \Downarrow \{((i, S), S').\ P\ i\ S\ S'\}\ (f\ S)$› **and**
‹*nofail gS*›
shows ‹$y \le \Downarrow \{((i, S), S').\ P\ i\ S\ S' \land Q\ S'\}\ (f\ S)$›
⟨*proof*⟩

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

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

lemma *refine-SPEC-refine-Down*:

‹$f \le SPEC\ C \longleftrightarrow f \le \Downarrow \{(T', T).\ T = T' \wedge C\ T'\}\ (SPEC\ C)$›
⟨*proof*⟩

0.0.5 More declarations

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

lemma *is-Nil-is-empty*[*sepref-fr-rules*]:
‹($return\ o\ is\text{-}Nil$, $RETURN\ o\ Multiset.is\text{-}empty$) $\in$ ($list\text{-}mset\text{-}assn\ R)^k \to_a bool\text{-}assn$›
⟨*proof*⟩

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

lemma *list-all2-remove*:
assumes
uniq: ‹$IS\text{-}RIGHT\text{-}UNIQUE\ (p2rel\ R)$› ‹$IS\text{-}LEFT\text{-}UNIQUE\ (p2rel\ R)$› **and**
Ra: ‹$R\ a\ aa$› **and**
all: ‹$list\text{-}all2\ R\ xs\ ys$›
shows
‹$\exists\, xs'.\ mset\ xs' = remove1\text{-}mset\ a\ (mset\ xs) \wedge$
$(\exists\, ys'.\ mset\ ys' = remove1\text{-}mset\ aa\ (mset\ ys) \wedge list\text{-}all2\ R\ xs'\ ys')$›
⟨*proof*⟩

lemma *remove1-remove1-mset*:
assumes *uniq*: ‹$IS\text{-}RIGHT\text{-}UNIQUE\ R$› ‹$IS\text{-}LEFT\text{-}UNIQUE\ R$›
shows ‹($uncurry\ (RETURN\ oo\ remove1)$, $uncurry\ (RETURN\ oo\ remove1\text{-}mset)$) $\in$
$R \times_r (list\text{-}mset\text{-}rel\ O\ \langle R\rangle\ mset\text{-}rel) \to_f$
$\langle list\text{-}mset\text{-}rel\ O\ \langle R\rangle\ mset\text{-}rel\rangle\ nres\text{-}rel$›
⟨*proof*⟩

lemma
Nil-list-mset-rel-iff:
‹$([], aaa) \in list\text{-}mset\text{-}rel \longleftrightarrow aaa = \{\#\}$› **and**
empty-list-mset-rel-iff:
‹$(a, \{\#\}) \in list\text{-}mset\text{-}rel \longleftrightarrow a = []$›
⟨*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 *snd-hnr-pure*:
‹$CONSTRAINT\ is\text{-}pure\ B \Longrightarrow (return \circ snd, RETURN \circ snd) \in A^d *_a B^k \to_a B$›
⟨*proof*⟩

0.0.6 List relation

lemma *list-rel-take*:
‹$(ba, ab) \in \langle A\rangle list\text{-}rel \Longrightarrow (take\ b\ ba, take\ b\ ab) \in \langle A\rangle list\text{-}rel$›
⟨*proof*⟩

lemma *list-rel-update'*:

fixes R
assumes *rel*: ‹$(xs, ys) \in \langle R\rangle$*list-rel*› **and**
h: ‹$(bi, b) \in R$›
shows ‹(*list-update xs ba bi*, *list-update ys ba b*) $\in \langle R\rangle$*list-rel*›
⟨*proof*⟩

lemma *list-rel-update*:
fixes R :: ‹$'a \Rightarrow 'b$ :: {*heap*}$\Rightarrow$ *assn*›
assumes *rel*: ‹$(xs, ys) \in \langle$*the-pure* $R\rangle$*list-rel*› **and**
h: ‹$h \models A * R\ b\ bi$› **and**
p: ‹*is-pure* R›
shows ‹(*list-update xs ba bi*, *list-update ys ba b*) $\in \langle$*the-pure* $R\rangle$*list-rel*›
⟨*proof*⟩

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

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

lemma *list-rel-mset-rel-def*[*refine-rel-defs*]:
‹$\langle R\rangle$*list-rel-mset-rel* $= \langle R\rangle$*list-rel O list-mset-rel*›
⟨*proof*⟩

lemma *list-mset-assn-pure-conv*:
‹*list-mset-assn* (*pure* R) = *pure* ($\langle R\rangle$*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 *list-rel-mset-rel-imp-same-length*: ‹$(a, b) \in \langle R\rangle$*list-rel-mset-rel* $\Longrightarrow$ *length* a = *size* b›
⟨*proof*⟩

0.0.7 More Functions, Relations, and Theorems

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

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

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

definition *emptied-arl* :: ‹$'a$ *array-list* $\Rightarrow$ $'a$ *array-list*› **where**
‹*emptied-arl* = $(\lambda(a, n).\ (a, 0))$›

lemma *emptied-arl-refine*[*sepref-fr-rules*]:
‹(*return o emptied-arl*, *RETURN o emptied-list*) $\in$ (*arl-assn* $R)^d \rightarrow_a$ *arl-assn* R›
⟨*proof*⟩

lemma *bool-assn-alt-def*: ‹*bool-assn* $a\ b = \uparrow (a = b)$›

$\langle proof \rangle$

lemma *nempty-list-mset-rel-iff*: ‹$M \neq \{\#\} \Longrightarrow$
$(xs,\ M) \in$ *list-mset-rel* $\longleftrightarrow (xs \neq [] \wedge hd\ xs \in\#\ M \wedge$
$(tl\ xs,\ remove1\text{-}mset\ (hd\ xs)\ M) \in list\text{-}mset\text{-}rel)$›
$\langle proof \rangle$

lemma *Down-itself-via-SPEC*:
assumes ‹$I \leq SPEC\ P$› **and** ‹$\bigwedge x.\ P\ x \Longrightarrow (x,\ x) \in R$›
shows ‹$I \leq \Downarrow R\ I$›
$\langle proof \rangle$

lemma *bind-if-inverse*:
‹*do* {
$S \leftarrow H$;
if b then f S else g S
} =
(*if b then do* $\{S \leftarrow H;\ f\ S\}$ *else do* $\{S \leftarrow H;\ g\ S\}$)
› **for** H :: ‹$'a\ nres$›
$\langle proof \rangle$

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

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

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

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

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 $(f,h) \in hfref\ P\ R'\ S'$
$\langle proof \rangle$

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* $= \{(c,\ b).\ c = ()\}$›

abbreviation *ghost-assn* **where**
‹$ghost\text{-}assn \equiv hr\text{-}comp\ unit\text{-}assn\ virtual\text{-}copy\text{-}rel$›

lemma [*sepref-fr-rules*]:
‹$(return\ o\ (\lambda\text{-}.\ ()),\ RETURN\ o\ virtual\text{-}copy) \in R^k \rightarrow_a ghost\text{-}assn$›
⟨*proof*⟩

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

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

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

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

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

lemma *foldli-cong2*:
assumes
le: ‹$length\ l = length\ l'$› **and**
σ: ‹$\sigma = \sigma'$› **and**
c: ‹$c = c'$› **and**
H: ‹$\bigwedge\sigma\ x.\ x < length\ l \Longrightarrow c'\ \sigma \Longrightarrow f\ (l\ !\ x)\ \sigma = f'\ (l'\ !\ x)\ \sigma$›
shows ‹$foldli\ l\ c\ f\ \sigma = foldli\ l'\ c'\ f'\ \sigma'$›
⟨*proof*⟩

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

lemma (**in** −) *WHILEIT-rule-stronger-inv-RES′*:
assumes
‹$wf\ R$› **and**
‹$I\ s$› **and**
‹$I'\ s$›
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \le SPEC\ (\lambda s'.\ I\ s' \wedge\ \ I'\ s' \wedge (s',\ s) \in R)$› **and**
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow \neg\ b\ s \Longrightarrow RETURN\ s \le \Downarrow H\ (RES\ \Phi)$›
shows ‹$WHILE_T{}^I\ b\ f\ s \le \Downarrow H\ (RES\ \Phi)$›
⟨*proof*⟩

lemma *RES-RES13-RETURN-RES*: ‹*do* {
  $(M, N, D, Q, W, vm, \varphi, clvls, cach, lbd, outl, stats, fast\text{-}ema, slow\text{-}ema, ccount,$
  $vdom, avdom, lcount) \leftarrow RES\ A;$
  $RES\ (f\ M\ N\ D\ Q\ W\ vm\ \varphi\ clvls\ cach\ lbd\ outl\ stats\ fast\text{-}ema\ slow\text{-}ema\ ccount$
  $vdom\ avdom\ lcount)$
} $= RES\ (\bigcup(M, N, D, Q, W, vm, \varphi, clvls, cach, lbd, outl, stats, fast\text{-}ema, slow\text{-}ema, ccount,$
  $vdom, avdom, lcount){\in}A.\ f\ M\ N\ D\ Q\ W\ vm\ \varphi\ clvls\ cach\ lbd\ outl\ stats\ fast\text{-}ema\ slow\text{-}ema\ ccount$
  $vdom\ avdom\ lcount)$›
  ⟨*proof*⟩

lemma *id-mset-list-assn-list-mset-assn*:
  assumes ‹*CONSTRAINT is-pure R*›
  shows ‹$(return\ o\ id, RETURN\ o\ mset) \in (list\text{-}assn\ R)^d \rightarrow_a list\text{-}mset\text{-}assn\ R$›
⟨*proof*⟩

lemma *RES-SPEC-conv*: ‹$RES\ P = SPEC\ (\lambda v.\ v \in P)$›
  ⟨*proof*⟩

0.0.8 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* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow nat \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a\ list$ *nres*› **where**
  ‹*insert-sort-inner R f xs i = do* {
    $(j, ys) \leftarrow WHILE_T{}^{\lambda(j, ys).\ j \geq 0 \land mset\ xs = mset\ ys \land j < length\ ys}$
      $(\lambda(j, ys).\ j > 0 \land R\ (f\ ys\ j)\ (f\ ys\ (j - 1)))$
      $(\lambda(j, ys).$ *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 *reorder-remove* :: ‹$'b \Rightarrow 'a\ list \Rightarrow 'a\ list\ nres$› **where**
‹$reorder\text{-}remove\ \text{-}\ removed = SPEC\ (\lambda removed'.\ mset\ removed' = mset\ removed)$›

definition *insert-sort* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow nat \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'a\ list\ nres$› **where**
  ‹*insert-sort R f xs = do* {
    $(i, ys) \leftarrow WHILE_T{}^{\lambda(i, ys).\ (ys = [] \lor i \leq length\ ys) \land mset\ xs = mset\ ys}$
      $(\lambda(i, ys).\ i < length\ ys)$
      $(\lambda(i, ys).$ *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*:
‹$(uncurry\ (insert\text{-}sort\text{-}inner\ R\ f),\ uncurry\ (\lambda m\ m'.\ reorder\text{-}remove\ m'\ m)) \in$
$[\lambda(xs,\ i).\ i < length\ xs]_f\ \langle Id{::}\ ('a \times 'a)\ set\rangle list\text{-}rel \times_r nat\text{-}rel \rightarrow \langle Id\rangle\ nres\text{-}rel$›
⟨*proof*⟩

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

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

definition ‹*op-arl-replicate* = *op-list-replicate*›
lemma *arl-fold-custom-replicate*:
‹*replicate* = *op-arl-replicate*›
⟨*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*]:
‹$(return\ o\ (\lambda(a,\ \text{-}).\ (a,\ 1{::}nat)),\ RETURN\ o\ take1) \in [\lambda xs.\ xs \neq []]_a\ (arl\text{-}assn\ R)^d \rightarrow arl\text{-}assn\ R$›
⟨*proof*⟩

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

abbreviation *uncurry4'* **where**
uncurry4' f ≡ *uncurry2* (*uncurry2 f*)

abbreviation *uncurry6'* **where**
uncurry6' f ≡ *uncurry2* (*uncurry4' f*)

lemma *Down-id-eq*: ⇓ *Id a* = *a*
⟨*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 ⟶ (
(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(λi.
(i ≠ None ⟶ P (C ! the i) ∧ the i ≥ a ∧ the i < b) ∧
(i = None ⟶ (∀j. j ≥ a ⟶ j < b ⟶ ¬P (C!j))))›
⟨*proof*⟩

end
theory *Array-Array-List*
imports *WB-More-Refinement*
begin

0.0.9 Array of Array Lists

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

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

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

definition *heap-list-all-nth*:: (′a ⇒ ′b ⇒ assn) ⇒ nat list ⇒ ′a list ⇒ ′b list ⇒ assn **where**
‹heap-list-all-nth R is xs ys = foldr ((∗)) (map (λi. 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 ‹$heap\text{-}list\text{-}all\text{-}nth\ R\ is\ xs\ ys = heap\text{-}list\text{-}all\text{-}nth\ R\ is'\ xs\ ys$›
  ⟨*proof*⟩

lemma *heap-list-add-same-length*:
  ‹$h \models heap\text{-}list\text{-}all\ R'\ xs\ p \Longrightarrow length\ p = length\ xs$›
  ⟨*proof*⟩

lemma *heap-list-all-nth-Suc*:
  assumes *a*: ‹$a > 1$›
  shows ‹$heap\text{-}list\text{-}all\text{-}nth\ R\ [Suc\ 0..<a]\ (x \# xs)\ (y \# ys) =$
    $heap\text{-}list\text{-}all\text{-}nth\ R\ [0..<a-1]\ xs\ ys$›
⟨*proof*⟩

lemma *heap-list-all-nth-append*:
  ‹$heap\text{-}list\text{-}all\text{-}nth\ R\ (is\ @\ is')\ xs\ ys = heap\text{-}list\text{-}all\text{-}nth\ R\ is\ xs\ ys * heap\text{-}list\text{-}all\text{-}nth\ R\ is'\ xs\ ys$›
  ⟨*proof*⟩

lemma *heap-list-all-heap-list-all-nth-eq*:
  ‹$heap\text{-}list\text{-}all\ R\ xs\ ys = heap\text{-}list\text{-}all\text{-}nth\ R\ [0..<\ length\ xs]\ xs\ ys * \uparrow(length\ xs = length\ ys)$›
  ⟨*proof*⟩

lemma *heap-list-all-nth-remove1*: ‹$i \in set\ is \Longrightarrow$
  $heap\text{-}list\text{-}all\text{-}nth\ R\ is\ xs\ ys = R\ (xs\ !\ i)\ (ys\ !\ i) * heap\text{-}list\text{-}all\text{-}nth\ R\ (remove1\ i\ is)\ xs\ ys$›
  ⟨*proof*⟩

definition *arrayO-assn* :: ‹$('a \Rightarrow 'b::heap \Rightarrow assn) \Rightarrow 'a\ list \Rightarrow 'b\ array \Rightarrow assn$› **where**
  ‹$arrayO\text{-}assn\ R'\ xs\ axs \equiv \exists_A\ p.\ array\text{-}assn\ id\text{-}assn\ p\ axs * heap\text{-}list\text{-}all\ R'\ xs\ p$›

definition *arrayO-except-assn*:: ‹$('a \Rightarrow 'b::heap \Rightarrow assn) \Rightarrow nat\ list \Rightarrow 'a\ list \Rightarrow 'b\ array \Rightarrow \text{-} \Rightarrow assn$›
where
  ‹$arrayO\text{-}except\text{-}assn\ R'\ is\ xs\ axs\ f \equiv$
    $\exists_A\ p.\ array\text{-}assn\ id\text{-}assn\ p\ axs * heap\text{-}list\text{-}all\text{-}nth\ R'\ (fold\ remove1\ is\ [0..<length\ xs])\ xs\ p *$
   $\uparrow (length\ xs = length\ p) * f\ p$›

lemma *arrayO-except-assn-array0*: ‹$arrayO\text{-}except\text{-}assn\ R\ []\ xs\ asx\ (\lambda\text{-}.\ emp) = arrayO\text{-}assn\ R\ xs\ asx$›
⟨*proof*⟩

lemma *arrayO-except-assn-array0-index*:
  ‹$i < length\ xs \Longrightarrow arrayO\text{-}except\text{-}assn\ R\ [i]\ xs\ asx\ (\lambda p.\ R\ (xs\ !\ i)\ (p\ !\ i)) = arrayO\text{-}assn\ R\ xs\ asx$›
  ⟨*proof*⟩

lemma *arrayO-nth-rule*[*sep-heap-rules*]:
  assumes *i*: ‹$i < length\ a$›
  shows ‹ $<arrayO\text{-}assn\ (arl\text{-}assn\ R)\ a\ ai>\ Array.nth\ ai\ i\ <\lambda r.\ arrayO\text{-}except\text{-}assn\ (arl\text{-}assn\ R)\ [i]\ a$
ai
  $(\lambda r'.\ arl\text{-}assn\ R\ (a\ !\ i)\ r * \uparrow(r = r'\ !\ i))>$›
⟨*proof*⟩

definition *length-a* :: ‹$'a::heap\ array \Rightarrow nat\ Heap$› **where**
  ‹$length\text{-}a\ xs = Array.len\ xs$›

lemma *length-a-rule*[*sep-heap-rules*]:
  ‹$<arrayO\text{-}assn\ R\ x\ xi>\ length\text{-}a\ xi\ <\lambda r.\ arrayO\text{-}assn\ R\ x\ xi * \uparrow(r = length\ x)>_t$›

⟨*proof*⟩

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

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

lemma *le-length-ll-nemptyD*: ‹$b < length\text{-}ll\ a\ ba \Longrightarrow a\ !\ ba \neq []$›
⟨*proof*⟩

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

lemma *length-aa-rule*[*sep-heap-rules*]:
‹$b < length\ xs \Longrightarrow <arrayO\text{-}assn\ (arl\text{-}assn\ R)\ xs\ a>\ length\text{-}aa\ a\ b$
$<\lambda r.\ arrayO\text{-}assn\ (arl\text{-}assn\ R)\ xs\ a * \uparrow (r = length\text{-}ll\ xs\ b)>_t$›
⟨*proof*⟩

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

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

lemma *models-heap-list-all-models-nth*:
‹$(h,\ as) \models heap\text{-}list\text{-}all\ R\ a\ b \Longrightarrow i < length\ a \Longrightarrow \exists as'.\ (h,\ as') \models R\ (a!i)\ (b!i)$›
⟨*proof*⟩

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

lemma *nth-aa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows
‹$(uncurry2\ nth\text{-}aa,\ uncurry2\ (RETURN \circ\circ\circ nth\text{-}ll)) \in$
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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

definition *append-ll* :: $'a\ list\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list\ list$ **where**

‹*append-ll xs i x* = *list-update xs i* (*xs* ! *i* @ [*x*])›

lemma *sep-auto-is-stupid*:
fixes *R* :: ‹′*a* ⇒ ′*b*::{*heap*,*default*} ⇒ *assn*›
assumes *p*: ‹*is-pure R*›
shows
‹<$\exists_A p$. *R1 p* ∗ *R2 p* ∗ *arl-assn R l′ aa* ∗ *R x x′* ∗ *R4 p*>
arl-append aa x′ <λ*r*. ($\exists_A p$. *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*)) ∈
$[\lambda((l,i),x).\ i < \mathit{length}\ l]_a$ (*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** ‹$ba < length\text{-}ll\ a\ bb$›
shows ‹$<R\ b\ bi * arrayO\text{-}assn\ (arl\text{-}assn\ R)\ a\ ai>\ update\text{-}aa\ ai\ bb\ ba\ bi$
$<\lambda r.\ R\ b\ bi * (\exists_A x.\ arrayO\text{-}assn\ (arl\text{-}assn\ R)\ x\ r * \uparrow (x = update\text{-}ll\ a\ bb\ ba\ b))>_t$›
⟨*proof*⟩

lemma *update-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹$(uncurry3\ update\text{-}aa,\ uncurry3\ (RETURN\ oooo\ update\text{-}ll)) \in$
$[\lambda(((l,i),\ j),\ x).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k *_a R^k \rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
⟨*proof*⟩

definition *set-butlast-ll* **where**
‹$set\text{-}butlast\text{-}ll\ xs\ i = xs[i := butlast\ (xs\ !\ i)]$›

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

lemma *list-rel-butlast*:
assumes *rel*: ‹$(xs,\ ys) \in \langle the\text{-}pure\ R\rangle list\text{-}rel$›
shows ‹$(butlast\ xs,\ butlast\ ys) \in \langle the\text{-}pure\ R\rangle list\text{-}rel$›
⟨*proof*⟩

lemma *arrayO-except-assn-arl-butlast*:
assumes ‹$b < length\ a$› **and**
‹$a\ !\ b \neq []$›
shows
‹$<arrayO\text{-}except\text{-}assn\ (arl\text{-}assn\ R)\ [b]\ a\ ai\ (\lambda r'.\ arl\text{-}assn\ R\ (a\ !\ b)\ (aa,\ ba) *$
$\uparrow ((aa,\ ba) = r'\ !\ b))>$
$arl\text{-}butlast\ (aa,\ ba)$
$<\lambda(aa,\ ba).\ arrayO\text{-}except\text{-}assn\ (arl\text{-}assn\ R)\ [b]\ a\ ai\ (\lambda r'.\ arl\text{-}assn\ R\ (butlast\ (a\ !\ b))\ (aa,\ ba)* true)>$›
⟨*proof*⟩

lemma *set-butlast-aa-rule*[*sep-heap-rules*]:
assumes ‹*is-pure R*› **and**
‹$b < length\ a$› **and**
‹$a\ !\ b \neq []$›
shows ‹$<arrayO\text{-}assn\ (arl\text{-}assn\ R)\ a\ ai>\ set\text{-}butlast\text{-}aa\ ai\ b$
$<\lambda r.\ (\exists_A x.\ arrayO\text{-}assn\ (arl\text{-}assn\ R)\ x\ r * \uparrow (x = set\text{-}butlast\text{-}ll\ a\ b))>_t$›
⟨*proof*⟩

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

definition *last-aa* :: $('a::heap\ array\text{-}list)\ array \Rightarrow nat \Rightarrow 'a\ Heap$ **where**
‹$last\text{-}aa\ xs\ i = do\ \{$

    $x \leftarrow$ *Array.nth xs i*;
    arl-last x
  }›

definition *last-ll* :: $'a$ *list list* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ **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* $\neq$ []›
  shows ‹
      <*arrayO-assn* (*arl-assn R*) *a ai*>
        last-aa ai b
      <$\lambda r.$ *arrayO-assn* (*arl-assn R*) *a ai* $*$ ($\exists_A x.$ *R x r* $*$ $\uparrow$ (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*)) $\in$
    $[\lambda(l,i).\ i <$ *length l* $\wedge$ *l* ! *i* $\neq$ $[]]_a$ (*arrayO-assn* (*arl-assn R*))^k $*_a$ *nat-assn*^{k} $\rightarrow$ *R*›
⟨*proof*⟩

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

lemma *nth-a-hnr*[*sepref-fr-rules*]:
  ‹(*uncurry nth-a*, *uncurry* (*RETURN oo op-list-get*)) $\in$
    $[\lambda(xs,\ i).\ i <$ *length xs*$]_a$ (*arrayO-assn* (*arl-assn R*))^k $*_a$ *nat-assn*^{k} $\rightarrow$ *arl-assn R*›
  ⟨*proof*⟩

definition *swap-aa* :: ($'a$::*heap array-list*) *array* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ($'a$ *array-list*) *array Heap*
where
  ‹*swap-aa xs k i j* = *do* {
    xi $\leftarrow$ *nth-aa xs k i*;
    xj $\leftarrow$ *nth-aa xs k j*;
    xs $\leftarrow$ *update-aa xs k i xj*;
    xs $\leftarrow$ *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
   <$\lambda r.\ \exists_A x.$ *arrayO-assn* (*arl-assn R*) *aa a* $*$
            (*R x r* $*$
            $\uparrow$ (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
      <λ*r*. $\exists_A$*x*. *invalid-assn* (*arrayO-assn* (*arl-assn R*)) *aa a* ∗ *arrayO-assn* (*arl-assn R*) *x r* ∗
         true ∗
         ↑ (*x* = *update-ll aa b ba be*)>›
⟨proof⟩

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

sepref-definition
  arrayO-ara-empty-sz-code
  is *RETURN o arrayO-ara-empty-sz*
  :: ‹$nat\text{-}assn^k \rightarrow_a arrayO\text{-}assn\ (arl\text{-}assn\ (R::'a \Rightarrow 'b::\{heap, default\} \Rightarrow assn))$›
  ⟨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 $\rightarrow_a$ *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*. ∃$_A$*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

0.0.10 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*:
‹$i < length\ xs \Longrightarrow arlO\text{-}assn\text{-}except\ R\ [i]\ xs\ asx\ (\lambda p.\ R\ (xs\ !\ i)\ (p\ !\ i)) = arlO\text{-}assn\ R\ xs\ asx$›
⟨*proof*⟩

lemma *arrayO-raa-nth-rule*[*sep-heap-rules*]:
assumes *i*: ‹$i < length\ a$›
shows ‹ $<arlO\text{-}assn\ (array\text{-}assn\ R)\ a\ ai>\ arl\text{-}get\ ai\ i <\lambda r.\ arlO\text{-}assn\text{-}except\ (array\text{-}assn\ R)\ [i]\ a\ ai$
$(\lambda r'.\ array\text{-}assn\ R\ (a\ !\ i)\ r * \uparrow(r = r'\ !\ i))>$›
⟨*proof*⟩

definition *length-ra* :: ‹$'a::heap\ arrayO\text{-}raa \Rightarrow nat\ Heap$› **where**
‹$length\text{-}ra\ xs = arl\text{-}length\ xs$›

lemma *length-ra-rule*[*sep-heap-rules*]:
‹$<arlO\text{-}assn\ R\ x\ xi>\ length\text{-}ra\ xi <\lambda r.\ arlO\text{-}assn\ R\ x\ xi * \uparrow(r = length\ x)>_t$›
⟨*proof*⟩

lemma *length-ra-hnr*[*sepref-fr-rules*]:
‹$(length\text{-}ra,\ RETURN\ o\ op\text{-}list\text{-}length) \in (arlO\text{-}assn\ R)^k \rightarrow_a nat\text{-}assn$›
⟨*proof*⟩

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

lemma *le-length-rll-nemptyD*: ‹$b < length\text{-}rll\ a\ ba \Longrightarrow a\ !\ ba \neq []$›
⟨*proof*⟩

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

lemma *length-raa-rule*[*sep-heap-rules*]:
‹$b < length\ xs \Longrightarrow <arlO\text{-}assn\ (array\text{-}assn\ R)\ xs\ a>\ length\text{-}raa\ a\ b$
$<\lambda r.\ arlO\text{-}assn\ (array\text{-}assn\ R)\ xs\ a * \uparrow (r = length\text{-}rll\ xs\ b)>_t$›
⟨*proof*⟩

lemma *length-raa-hnr*[*sepref-fr-rules*]: ‹$(uncurry\ length\text{-}raa,\ uncurry\ (RETURN \circ\circ length\text{-}rll)) \in$
$[\lambda(xs,\ i).\ i < length\ xs]_a\ (arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow nat\text{-}assn$›
⟨*proof*⟩

definition *nth-raa* :: ‹$'a::heap\ arrayO\text{-}raa \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ Heap$› **where**
‹$nth\text{-}raa\ xs\ i\ j = do\ \{$
$x \leftarrow arl\text{-}get\ xs\ i;$
$y \leftarrow Array.nth\ x\ j;$
$return\ y\}$›

definition *nth-rll* :: $'a\ list\ list \Rightarrow nat \Rightarrow nat \Rightarrow 'a$ **where**
‹$nth\text{-}rll\ l\ i\ j = l\ !\ i\ !\ j$›

lemma *nth-raa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹$is\text{-}pure\ R$›
shows
‹$(uncurry2\ nth\text{-}raa,\ uncurry2\ (RETURN \circ\circ\circ nth\text{-}rll)) \in$
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›

$\langle proof \rangle$

definition *update-raa* :: $('a::\{heap,default\})$ *arrayO-raa* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$ *arrayO-raa Heap* **where**

‹*update-raa a i j y = do* {
  $x \leftarrow$ *arl-get a i*;
  $a' \leftarrow$ *Array.upd j y x*;
  arl-set a i a'
}› — is the Array.upd really needed?

definition *update-rll* :: $'a$ *list-rll* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$ *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 $\uparrow ((aa,\ bc) = r'\ !\ bb)$

lemma *arlO-assn-except-arl-set*[*sep-heap-rules*]:
fixes R :: ‹$'a \Rightarrow 'b$ :: {*heap*} $\Rightarrow$ *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* ($\lambda r'$. *array-assn R* (*a* ! *bb*) *aa* $*$
$\uparrow$ ($aa = r'$! *bb*)) $*$ *R b bi*$>$
Array.upd ba bi aa
$<\lambda aa$. *arlO-assn-except* (*array-assn R*) [*bb*] *a ai*
($\lambda r'$. *array-assn R* ((*a* ! *bb*)[*ba* := *b*]) *aa*) $*$ *R b bi* $*$ *true*$>$›
$\langle proof \rangle$

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*
$<\lambda r$. *R b bi* $*$ ($\exists_A x$. *arlO-assn* (*array-assn R*) *x r* $*$ $\uparrow$ ($x =$ *update-rll a bb ba b*))$>_t$›
$\langle proof \rangle$

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

definition *swap-aa* :: $('a::\{heap,default\})$ *arrayO-raa* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *arrayO-raa Heap* **where**

‹*swap-aa xs k i j = do* {
  $xi \leftarrow$ *nth-raa xs k i*;
  $xj \leftarrow$ *nth-raa xs k j*;
  $xs \leftarrow$ *update-raa xs k i xj*;
  $xs \leftarrow$ *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
<$\lambda r.\ \exists_A x.$ *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
<$\lambda r.\ \exists_A x.$ *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*)) ∈
$[\lambda(((xs,\ k),\ i),\ j).\ k < \mathit{length}\ xs \wedge i < \mathit{length\text{-}rll}\ xs\ k \wedge j < \mathit{length\text{-}rll}\ xs\ k]_a$
$(\mathit{arlO\text{-}assn}\ (\mathit{array\text{-}assn}\ R))^d *_a \mathit{nat\text{-}assn}^k *_a \mathit{nat\text{-}assn}^k *_a \mathit{nat\text{-}assn}^k \rightarrow (\mathit{arlO\text{-}assn}\ (\mathit{array\text{-}assn}\ R))$›
⟨*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*)) ∈
$[\lambda((xs,\ n),\ \text{-}).\ n < \mathit{length}\ xs]_a\ (\mathit{arlO\text{-}assn}\ R)^d *_a \mathit{nat\text{-}assn}^k *_a R^d \rightarrow (\mathit{arlO\text{-}assn}\ R)$›
⟨*proof*⟩
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 $\leftarrow$ *Array.new 0 default*;
$a \leftarrow$ *array-grow a newcap default*;
$a \leftarrow$ *Array.upd n x a*;
return $(a,n+1)$
$\}$
$\}$

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

lemma *heap-list-all-Nil-append*:
‹$y \neq [] \Longrightarrow$ *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* <$\lambda a.$ *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*)) $\in$
$(arlO\text{-}assn\ R)^d *_a R^d \rightarrow_a arlO\text{-}assn\ R$›
⟨*proof*⟩

definition *array-of-arl* :: ‹$'a$ *list* $\Rightarrow$ $'a$ *list*› **where**
‹*array-of-arl xs* $=$ *xs*›

definition *array-of-arl-raa* :: $'a$::*heap array-list* $\Rightarrow$ $'a$ *array Heap* **where**
‹*array-of-arl-raa* $= (\lambda(a, n).$ *array-shrink a n*$)$›

lemma *array-of-arl*[*sepref-fr-rules*]:
‹(*array-of-arl-raa*, *RETURN o array-of-arl*) $\in (arl\text{-}assn\ R)^d \rightarrow_a (array\text{-}assn\ R)$›
⟨*proof*⟩

definition *arrayO-raa-empty* $\equiv$ *do* $\{$
$a \leftarrow$ *Array.new initial-capacity default*;
return $(a,0)$
$\}$

lemma *arrayO-raa-empty-rule*[*sep-heap-rules*]: < *emp* > *arrayO-raa-empty* <$\lambda r.$ *arlO-assn* R $[]$ r>
⟨*proof*⟩

definition *arrayO-raa-empty-sz* **where**
arrayO-raa-empty-sz init-cap $\equiv$ *do* $\{$
default $\leftarrow$ *Array.new 0 default*;
$a \leftarrow$ *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 <$\lambda r.$ *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*)) ∈
$[\lambda(xs,\ n).\ n < length\ xs]_a\ (arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow array\text{-}assn\ R$›
⟨*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*) ∈
$[\lambda xs.\ xs \neq []]_a\ (array\text{-}assn\ R)^d \rightarrow (arl\text{-}assn\ R)$›
⟨*proof*⟩

end
theory *WB-Word-Assn*
imports
HOL−Word.Word
Bits-Natural
WB-More-Refinement
Native-Word.Uint64
begin

0.0.11 More Setup for Fixed Size Natural Numbers

Words

lemma *less-upper-bintrunc-id*: ‹$n < 2 \,\hat{}\, b \Longrightarrow n \geq 0 \Longrightarrow bintrunc\ b\ n = n$›
⟨*proof*⟩

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

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

lemma *bintrunc-eq-bits-eqI*: ‹ $(\bigwedge n.\ (n < r \wedge bin\text{-}nth\ c\ n) = (n < r \wedge bin\text{-}nth\ a\ n)) \Longrightarrow$
$bintrunc\ r\ (a) = bintrunc\ r\ c$›
⟨*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**
‹$uint32\text{-}max = 2\ \hat{}32 - 1$›

lemma *unat-le-uint32-max-no-bit-set*:
fixes *n* :: ‹*'a::len word*›
assumes *less*: ‹$unat\ n \leq uint32\text{-}max$› **and**
n: ‹*n* !! *na*› **and**
32: ‹$32 < LENGTH('a)$›
shows ‹$na < 32$›
⟨*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:
‹$unat\ n \leq uint32\text{-}max \Longrightarrow$ ∃ *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** ‹$- \Longrightarrow ?B$›) **and**
ex-rbl-word64-ge-uint32-max:
‹$n\ AND\ (2\hat{}32 - 1) = 0 \Longrightarrow$ ∃ *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*)›

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*⟩

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*›

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

lemma *op-eq-uint32*:
‹(*uncurry* (*return oo* ((=) :: *uint32* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈
$uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a bool\text{-}assn$›
⟨*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*)) ∈
$uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
⟨*proof*⟩

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

lemma *nat-of-uint32-0-iff*: ‹$nat\text{-}of\text{-}uint32\ xi = 0 \longleftrightarrow xi = 0$› **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**
  ‹$uint32\text{-}safe\text{-}minus\ m\ n = (if\ m < n\ then\ 0\ else\ m - n)$›

lemma *nat-of-uint32-le-minus*: ‹$ai \leq bi \Longrightarrow 0 = nat\text{-}of\text{-}uint32\ ai - nat\text{-}of\text{-}uint32\ bi$›
  ⟨*proof*⟩

lemma *nat-of-uint32-notle-minus*:
  ‹$\neg\ ai < bi \Longrightarrow$
    $nat\text{-}of\text{-}uint32\ (ai - bi) = nat\text{-}of\text{-}uint32\ ai - nat\text{-}of\text{-}uint32\ bi$›
  ⟨*proof*⟩

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

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

lemma *nat-of-uint32-uint32-of-nat-id*: ‹$n \leq uint32\text{-}max \Longrightarrow nat\text{-}of\text{-}uint32\ (uint32\text{-}of\text{-}nat\ n) = n$›
  ⟨*proof*⟩

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

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

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

lemma *uint32-less-than-0*[*iff*]: ‹$(a::uint32) \leq 0 \longleftrightarrow a = 0$›
  ⟨*proof*⟩

lemma *nat-of-uint32-less-iff*: ‹$nat\text{-}of\text{-}uint32\ a < nat\text{-}of\text{-}uint32\ b \longleftrightarrow a < b$›
⟨*proof*⟩

lemma *nat-of-uint32-le-iff*: ‹$nat\text{-}of\text{-}uint32\ a \leq nat\text{-}of\text{-}uint32\ b \longleftrightarrow a \leq b$›
⟨*proof*⟩

lemma *nat-of-uint32-max*:
‹$nat\text{-}of\text{-}uint32\ (max\ ai\ bi) = max\ (nat\text{-}of\text{-}uint32\ ai)\ (nat\text{-}of\text{-}uint32\ bi)$›
⟨*proof*⟩

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

lemma *nat-of-uint32-distrib-mult2*:
assumes ‹$nat\text{-}of\text{-}uint32\ xi \leq uint32\text{-}max\ div\ 2$›
shows ‹$nat\text{-}of\text{-}uint32\ (2 * xi) = 2 * nat\text{-}of\text{-}uint32\ xi$›
⟨*proof*⟩

lemma *nat-of-uint32-distrib-mult2-plus1*:
assumes ‹$nat\text{-}of\text{-}uint32\ xi \leq uint32\text{-}max\ div\ 2$›
shows ‹$nat\text{-}of\text{-}uint32\ (2 * xi + 1) = 2 * nat\text{-}of\text{-}uint32\ xi + 1$›
⟨*proof*⟩

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

lemma *array-set-hnr-u*:
‹$CONSTRAINT\ is\text{-}pure\ A \Longrightarrow$
$(uncurry2\ (\lambda xs\ i.\ heap\text{-}array\text{-}set\ xs\ (nat\text{-}of\text{-}uint32\ i)),\ uncurry2\ (RETURN \circ\circ\circ op\text{-}list\text{-}set)) \in$
$[pre\text{-}list\text{-}set]_a\ (array\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \rightarrow array\text{-}assn\ A$›
⟨*proof*⟩

lemma *array-get-hnr-u*:
assumes ‹$CONSTRAINT\ is\text{-}pure\ A$›
shows ‹$(uncurry\ (\lambda xs\ i.\ Array.nth\ xs\ (nat\text{-}of\text{-}uint32\ i)),$
$uncurry\ (RETURN \circ\circ op\text{-}list\text{-}get)) \in [pre\text{-}list\text{-}get]_a\ (array\text{-}assn\ A)^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

lemma *arl-get-hnr-u*:
assumes ‹$CONSTRAINT\ is\text{-}pure\ A$›
shows ‹$(uncurry\ (\lambda xs\ i.\ arl\text{-}get\ xs\ (nat\text{-}of\text{-}uint32\ i)),\ uncurry\ (RETURN \circ\circ op\text{-}list\text{-}get))$
$\in [pre\text{-}list\text{-}get]_a\ (arl\text{-}assn\ A)^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

lemma *nat-of-uint32-add*:
‹$nat\text{-}of\text{-}uint32\ ai + nat\text{-}of\text{-}uint32\ bi \leq uint32\text{-}max \Longrightarrow$
$nat\text{-}of\text{-}uint32\ (ai + bi) = nat\text{-}of\text{-}uint32\ ai + nat\text{-}of\text{-}uint32\ bi$›
⟨*proof*⟩

lemma *uint32-nat-assn-plus*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (+)),\ uncurry\ (RETURN\ oo\ (+))) \in [\lambda(m,\ n).\ m + n \leq uint32\text{-}max]_a$

$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn$›
⟨proof⟩

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

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

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

definition *zero-uint32-nat* **where**
[*simp*]: ‹$zero\text{-}uint32\text{-}nat = (0 :: nat)$›

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

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

definition *one-uint32-nat* **where**
[*simp*]: ‹$one\text{-}uint32\text{-}nat = (1 :: nat)$›

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

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

definition *two-uint32-nat* **where** [*simp*]: ‹$two\text{-}uint32\text{-}nat = (2 :: nat)$›

definition *two-uint32* **where**
[*simp*]: ‹$two\text{-}uint32 = (2 :: uint32)$›

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

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

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

lemma *le-nat-uint32-hnr*:

‹(*uncurry* (*return oo* (λ*a b*. *a* < *nat-of-uint32 b*)), *uncurry* (*RETURN oo* (<))) ∈ *nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

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*⟩

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*)) ∈ [λ(*m*, *n*). *m* ≥ *n*]$_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} → *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*)) ∈ [λ(*m*, *n*). *m* ≥ *n*]$_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} → *uint32-nat-assn*›
⟨*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*⟩

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

lemma *uint32-nat-assn-nat-assn-nat-of-uint32*:
‹*uint32-nat-assn aa a* = *nat-assn aa* (*nat-of-uint32 a*)›
⟨*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*⟩

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

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

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

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

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

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

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

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*)›

$\langle proof \rangle$

lemma *int-of-uint32-numeral*[*simp*]:
$\langle numeral\ n \le ((2\ \widehat{}\ 32 - 1)::nat) \Longrightarrow int\text{-}of\text{-}uint32\ (numeral\ n) = numeral\ n\rangle$
$\langle proof \rangle$

lemma *nat-of-uint32-numeral-iff*[*simp*]:
$\langle numeral\ n \le ((2\ \widehat{}\ 32 - 1)::nat) \Longrightarrow nat\text{-}of\text{-}uint32\ a = numeral\ n \longleftrightarrow a = numeral\ n\rangle$
$\langle proof \rangle$

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

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

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

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

lemma *nat-of-uint32-mult-le*:
$\langle nat\text{-}of\text{-}uint32\ ai * nat\text{-}of\text{-}uint32\ bi \le uint32\text{-}max \Longrightarrow$
$nat\text{-}of\text{-}uint32\ (ai * bi) = nat\text{-}of\text{-}uint32\ ai * nat\text{-}of\text{-}uint32\ bi\rangle$
$\langle proof \rangle$

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

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*⟩

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*⟩

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

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

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

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

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

lemma *word-nat-of-uint64-Rep-inject*[*simp*]: ‹*nat-of-uint64 ai* = *nat-of-uint64 bi* ⟷ *ai* = *bi*›
⟨*proof*⟩

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

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

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

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*⟩

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

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

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

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

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

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

lemma *nat-of-uint64-distrib-mult2*:
assumes ‹*nat-of-uint64 xi* ≤ *uint64-max div 2*›

shows ‹$nat\text{-}of\text{-}uint64\ (2 * xi) = 2 * nat\text{-}of\text{-}uint64\ xi$›
⟨*proof*⟩

lemma (**in** −)*nat-of-uint64-distrib-mult2-plus1*:
assumes ‹$nat\text{-}of\text{-}uint64\ xi \leq uint64\text{-}max\ div\ 2$›
shows ‹$nat\text{-}of\text{-}uint64\ (2 * xi + 1) = 2 * nat\text{-}of\text{-}uint64\ xi + 1$›
⟨*proof*⟩

lemma *nat-of-uint64-numeral*[*simp*]:
‹$numeral\ n \leq ((2\ \hat{}\ 64 - 1)::nat) \Longrightarrow nat\text{-}of\text{-}uint64\ (numeral\ n) = numeral\ n$›
⟨*proof*⟩

lemma *int-of-uint64-alt-def*: ‹$int\text{-}of\text{-}uint64\ n = int\ (nat\text{-}of\text{-}uint64\ n)$›
⟨*proof*⟩

lemma *int-of-uint64-numeral*[*simp*]:
‹$numeral\ n \leq ((2\ \hat{}\ 64 - 1)::nat) \Longrightarrow int\text{-}of\text{-}uint64\ (numeral\ n) = numeral\ n$›
⟨*proof*⟩

lemma *nat-of-uint64-numeral-iff*[*simp*]:
‹$numeral\ n \leq ((2\ \hat{}\ 64 - 1)::nat) \Longrightarrow nat\text{-}of\text{-}uint64\ a = numeral\ n \longleftrightarrow a = numeral\ n$›
⟨*proof*⟩

lemma *numeral-uint64-eq-iff*[*simp*]:
‹$numeral\ m \leq (2\hat{}64-1 :: nat) \Longrightarrow numeral\ n \leq (2\hat{}64-1 :: nat) \Longrightarrow ((numeral\ m :: uint64) = numeral\ n) \longleftrightarrow numeral\ m = (numeral\ n :: nat)$›
⟨*proof*⟩

lemma *numeral-uint64-eq0-iff*[*simp*]:
‹$numeral\ n \leq (2\hat{}64-1 :: nat) \Longrightarrow ((0 :: uint64) = numeral\ n) \longleftrightarrow 0 = (numeral\ n :: nat)$›
⟨*proof*⟩

lemma *transfer-pow-uint64*: ‹$Transfer.Rel\ (rel\text{-}fun\ cr\text{-}uint64\ (rel\text{-}fun\ (=)\ cr\text{-}uint64))\ (\hat{})\ (\hat{})$›
⟨*proof*⟩

lemma *shiftl-t2n-uint64*: ‹$n << m = n * 2\ \hat{}\ m$› **for** $n :: uint64$
⟨*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 *mod2-bin-last*: ‹a *mod* $2 = 0 \longleftrightarrow \neg$*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* $x \leq$ *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*:
  ‹$ai \geq 0 \Longrightarrow bi \geq 0 \Longrightarrow$ *nat* (ai *AND* bi) = *nat* ai *AND* *nat* bi›

$\langle proof \rangle$

lemma *nat-of-uint64-and*:
$\langle$*nat-of-uint64 ai* $\leq$ *uint64-max* $\Longrightarrow$ *nat-of-uint64 bi* $\leq$ *uint64-max* $\Longrightarrow$
nat-of-uint64 (*ai AND bi*) = *nat-of-uint64 ai AND nat-of-uint64 bi*$\rangle$
$\langle proof \rangle$

lemma *bitAND-uint64-max-hnr*[*sepref-fr-rules*]:
$\langle$(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*)))
$\in [\lambda(a, b).\ a \leq uint64\text{-}max \wedge b \leq uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn\rangle$
$\langle proof \rangle$

definition *two-uint64-nat* :: *nat* **where**
[*simp*]: $\langle$*two-uint64-nat* = *2*$\rangle$

lemma *two-uint64-nat*[*sepref-fr-rules*]:
$\langle$(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint64-nat*))
$\in\ unit\text{-}assn^k \rightarrow_a uint64\text{-}nat\text{-}assn\rangle$
$\langle proof \rangle$

lemma *nat-or*:
$\langle$*ai* $\geq$ *0* $\Longrightarrow$ *bi* $\geq$ *0* $\Longrightarrow$ *nat* (*ai OR bi*) = *nat ai OR nat bi*$\rangle$
$\langle proof \rangle$

lemma *nat-of-uint64-or*:
$\langle$*nat-of-uint64 ai* $\leq$ *uint64-max* $\Longrightarrow$ *nat-of-uint64 bi* $\leq$ *uint64-max* $\Longrightarrow$
nat-of-uint64 (*ai OR bi*) = *nat-of-uint64 ai OR nat-of-uint64 bi*$\rangle$
$\langle proof \rangle$

lemma *bitOR-uint64-max-hnr*[*sepref-fr-rules*]:
$\langle$(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*)))
$\in [\lambda(a, b).\ a \leq uint64\text{-}max \wedge b \leq uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn\rangle$
$\langle proof \rangle$

lemma *Suc-0-le-uint64-max*: $\langle$*Suc 0* $\leq$ *uint64-max*$\rangle$
$\langle proof \rangle$

lemma *nat-of-uint64-le-iff*: $\langle$*nat-of-uint64 a* $\leq$ *nat-of-uint64 b* $\longleftrightarrow$ *a* $\leq$ *b*$\rangle$
$\langle proof \rangle$

lemma *nat-of-uint64-notle-minus*:
$\langle\neg$ *ai* < *bi* $\Longrightarrow$
nat-of-uint64 (*ai* − *bi*) = *nat-of-uint64 ai* − *nat-of-uint64 bi*$\rangle$
$\langle proof \rangle$

lemma *fast-minus-uint64-nat*[*sepref-fr-rules*]:
$\langle$(*uncurry* (*return oo fast-minus*), *uncurry* (*RETURN oo fast-minus*))
$\in [\lambda(a, b).\ a \geq b]_a\ uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn\rangle$
$\langle proof \rangle$

lemma *fast-minus-uint64*[*sepref-fr-rules*]:
$\langle$(*uncurry* (*return oo fast-minus*), *uncurry* (*RETURN oo fast-minus*))
$\in [\lambda(a, b).\ a \geq b]_a\ uint64\text{-}assn^k *_a uint64\text{-}assn^k \rightarrow uint64\text{-}assn\rangle$

$\langle proof \rangle$

lemma *le-uint32-max-le-uint64-max*: ‹$a \leq$ *uint32-max* + *2* $\Longrightarrow a \leq$ *uint64-max*›
$\langle proof \rangle$

lemma *nat-of-uint64-ge-minus*:
‹$ai \geq bi \Longrightarrow$
nat-of-uint64 $(ai - bi) =$ *nat-of-uint64* $ai -$ *nat-of-uint64* bi›
$\langle proof \rangle$

lemma *minus-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (−)), *uncurry* (*RETURN oo* (−))) ∈
$[\lambda(a, b).\ a \geq b]_a$ *uint64-nat-assn*$^k *_a$ *uint64-nat-assn*$^k \rightarrow$ *uint64-nat-assn*›
$\langle proof \rangle$

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*›
$\langle proof \rangle$

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*›
$\langle proof \rangle$

lemma *sum-mod-uint64-max-le-uint64-max*[*simp*]: ‹*sum-mod-uint64-max a b* ≤ *uint64-max*›
$\langle proof \rangle$

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*›
$\langle proof \rangle$

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*)›

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

definition *zero-uint32* **where**
‹*zero-uint32* ≡ (*0* :: *uint32*)›

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*⟩

definition *two-uint64* **where** ‹*two-uint64* = (*2* :: *uint64*)›

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} $\rightarrow_a$ *uint32-assn*›
⟨*proof*⟩

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

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*⟩

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} $\rightarrow_a$ *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} $\rightarrow_a$ *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} $\rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *bitOR-uint64-assn*[*sepref-fr-rules*]:

‹$(uncurry\ (return\ oo\ (OR)),\ uncurry\ (RETURN\ oo\ (OR))) \in$
$uint64\text{-}assn^k *_a uint64\text{-}assn^k \rightarrow_a uint64\text{-}assn$›
⟨proof⟩

lemma *nat-of-uint64-mult-le*:
‹$nat\text{-}of\text{-}uint64\ ai * nat\text{-}of\text{-}uint64\ bi \leq uint64\text{-}max \Longrightarrow$
$nat\text{-}of\text{-}uint64\ (ai * bi) = nat\text{-}of\text{-}uint64\ ai * nat\text{-}of\text{-}uint64\ bi$›
⟨proof⟩

lemma *uint64-nat-assn-mult*:
‹$(uncurry\ (return\ oo\ ((*))),\ uncurry\ (RETURN\ oo\ ((*)))) \in [\lambda(a,\ b).\ a * b \leq uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨proof⟩

lemma *uint64-max-uint64-nat-assn*:
‹$(uncurry0\ (return\ 18446744073709551615),\ uncurry0\ (RETURN\ uint64\text{-}max)) \in$
$unit\text{-}assn^k \rightarrow_a uint64\text{-}nat\text{-}assn$›
⟨proof⟩

lemma *uint64-max-nat-assn*[*sepref-fr-rules*]:
‹$(uncurry0\ (return\ 18446744073709551615),\ uncurry0\ (RETURN\ uint64\text{-}max)) \in$
$unit\text{-}assn^k \rightarrow_a nat\text{-}assn$›
⟨proof⟩

lemma *bit-lshift-uint64-assn*:
‹$(uncurry\ (return\ oo\ (>>)),\ uncurry\ (RETURN\ oo\ (>>))) \in$
$uint64\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a uint64\text{-}assn$›
⟨proof⟩

Conversions

From nat to 64 bits **definition** *uint64-of-nat-conv* :: ‹$nat \Rightarrow nat$› **where**
‹$uint64\text{-}of\text{-}nat\text{-}conv\ i = i$›

lemma *uint64-of-nat-conv-hnr*[*sepref-fr-rules*]:
‹$(return\ o\ uint64\text{-}of\text{-}nat,\ RETURN\ o\ uint64\text{-}of\text{-}nat\text{-}conv) \in$
$[\lambda n.\ n \leq uint64\text{-}max]_a\ nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨proof⟩

From nat to 32 bits **definition** *nat-of-uint32-spec* :: ‹$nat \Rightarrow nat$› **where**
[*simp*]: ‹$nat\text{-}of\text{-}uint32\text{-}spec\ n = n$›

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

From 64 to nat bits **definition** *nat-of-uint64-conv* :: ‹$nat \Rightarrow nat$› **where**
[*simp*]: ‹$nat\text{-}of\text{-}uint64\text{-}conv\ i = i$›

lemma *nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
‹$(return\ o\ nat\text{-}of\text{-}uint64,\ RETURN\ o\ nat\text{-}of\text{-}uint64\text{-}conv) \in uint64\text{-}nat\text{-}assn^k \rightarrow_a nat\text{-}assn$›
⟨proof⟩

lemma *nat-of-uint64*[*sepref-fr-rules*]:
‹$(return\ o\ nat\text{-}of\text{-}uint64,\ RETURN\ o\ nat\text{-}of\text{-}uint64) \in$

$(uint64\text{-}assn)^k \rightarrow_a nat\text{-}assn$›
⟨proof⟩

From 32 to nat bits **definition** *nat-of-uint32-conv* :: ‹$nat \Rightarrow nat$› **where**
[*simp*]: ‹$nat\text{-}of\text{-}uint32\text{-}conv\ i = i$›

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

definition *convert-to-uint32* :: ‹$nat \Rightarrow nat$› **where**
[*simp*]: ‹$convert\text{-}to\text{-}uint32 = id$›

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

From 32 to 64 bits **definition** *uint64-of-uint32-conv* :: ‹$nat \Rightarrow nat$› **where**
[*simp*]: ‹$uint64\text{-}of\text{-}uint32\text{-}conv\ x = x$›

lemma *nat-of-uint32-le-uint32-max*: ‹$nat\text{-}of\text{-}uint32\ n \leq uint32\text{-}max$›
⟨proof⟩

lemma *nat-of-uint32-le-uint64-max*: ‹$nat\text{-}of\text{-}uint32\ n \leq uint64\text{-}max$›
⟨proof⟩

lemma *nat-of-uint64-uint64-of-uint32*: ‹$nat\text{-}of\text{-}uint64\ (uint64\text{-}of\text{-}uint32\ n) = nat\text{-}of\text{-}uint32\ n$›
⟨proof⟩

lemma *uint64-of-uint32-hnr*[*sepref-fr-rules*]:
‹$(return\ o\ uint64\text{-}of\text{-}uint32,\ RETURN\ o\ uint64\text{-}of\text{-}uint32) \in uint32\text{-}assn^k \rightarrow_a uint64\text{-}assn$›
⟨proof⟩

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

From 64 to 32 bits **definition** *uint32-of-uint64* **where**
‹$uint32\text{-}of\text{-}uint64\ n = uint32\text{-}of\text{-}nat\ (nat\text{-}of\text{-}uint64\ n)$›

definition *uint32-of-uint64-conv* **where**
[*simp*]: ‹$uint32\text{-}of\text{-}uint64\text{-}conv\ n = n$›

lemma *uint32-of-uint64-conv-hnr*[*sepref-fr-rules*]:
‹$(return\ o\ uint32\text{-}of\text{-}uint64,\ RETURN\ o\ uint32\text{-}of\text{-}uint64\text{-}conv) \in$
$[\lambda a.\ a \leq uint32\text{-}max]_a\ uint64\text{-}nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn$›
⟨proof⟩

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

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

0.0.12 More about general arrays

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

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

lemma *butlast-arl-hnr*[*sepref-fr-rules*]:
  ‹(*return o butlast-arl*, *RETURN o butlast*) ∈ $[\lambda xs.\ xs \neq []]_a$ $(arl\text{-}assn\ A)^d \rightarrow arl\text{-}assn\ A$›
⟨*proof*⟩

0.0.13 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*]:
  assumes ‹*CONSTRAINT is-pure R*›
  shows
    ‹(*uncurry2 nth-aa-u*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
      $[\lambda((x, L), L'). L < \mathit{length}\ x \wedge L' < \mathit{length}\ (x\ !\ L)]_a$
      $(\mathit{arrayO\text{-}assn}\ (\mathit{arl\text{-}assn}\ R))^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k *_a \mathit{nat\text{-}assn}^k \rightarrow R$›
  ⟨*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*)) ∈
      $[\lambda((l,i),j). i < \mathit{length}\ l \wedge j < \mathit{length\text{-}rll}\ l\ i]_a$
      $(\mathit{arlO\text{-}assn}\ (\mathit{array\text{-}assn}\ R))^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k *_a \mathit{nat\text{-}assn}^k \rightarrow R$›
  ⟨*proof*⟩

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

lemma *array-get-hnr-u*[*sepref-fr-rules*]:
  assumes ‹*CONSTRAINT is-pure A*›
  shows ‹(*uncurry nth-u-code*,
     uncurry (*RETURN* ∘∘ *op-list-get*)) ∈ $[\mathit{pre\text{-}list\text{-}get}]_a\ (\mathit{array\text{-}assn}\ A)^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k \rightarrow A$›
⟨*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 ≡ λ*a i*. *arl-get′ a* (*integer-of-uint32 i*)

lemma *arrayO-arl-get-u-rule*[*sep-heap-rules*]:
  assumes *i*: ‹*i* < *length a*› **and** ‹(*i′* , *i*) ∈ *uint32-nat-rel*›
  shows ‹<*arlO-assn* (*array-assn R*) *a ai*> *arl-get-u ai i′* <λ*r*. *arlO-assn-except* (*array-assn R*) [*i*] *a ai*
   (λ*r′*. *array-assn R* (*a* ! *i*) *r* ∗ ↑(*r* = *r′* ! *i*))>›

⟨*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*))
∈ $[\textit{pre-list-get}]_a$ $(\textit{arl-assn } A)^k *_a \textit{uint32-nat-assn}^k \rightarrow A$›
⟨*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*)) ∈
$[\lambda((l,i),j).\ i < \textit{length } l \wedge j < \textit{length-rll } l\ i]_a$
$(\textit{arlO-assn } (\textit{array-assn } R))^k *_a \textit{nat-assn}^k *_a \textit{uint32-nat-assn}^k \rightarrow R$›
⟨*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*)) ∈
$[\lambda((l,i),j).\ i < \textit{length } l \wedge j < \textit{length-rll } l\ i]_a$
$(\textit{arrayO-assn } (\textit{arl-assn } R))^k *_a \textit{uint64-nat-assn}^k *_a \textit{uint32-nat-assn}^k \rightarrow R$›
⟨*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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*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*)) ∈
$[\lambda(xs,\ n).\ nat\text{-}of\text{-}uint64\ n < length\ xs]_a\ (array\text{-}assn\ A)^k *_a uint64\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

lemma *array-get-hnr-u64*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry nth-u64-code*,
uncurry (*RETURN* ∘∘ *op-list-get*)) ∈ $[pre\text{-}list\text{-}get]_a\ (array\text{-}assn\ A)^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*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*)) ∈
$[pre\text{-}list\text{-}set]_a\ (array\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \rightarrow array\text{-}assn\ A$›
⟨*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′ ← *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*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(arl\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \to arl\text{-}assn\ A$›
⟨*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*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(array\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a A^k \to array\text{-}assn\ A$›
⟨*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*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(arl\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a A^k \to arl\text{-}assn\ A$›
⟨*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*)) ∈
      [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
      (*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *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-list-get*]$_a$ (*arl-assn A*)^k $*_a$ *uint64-nat-assn*^{k} → *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*)) ∈
      [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
      (*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *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*)) ∈
      [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
      (*arlO-assn* (*array-assn R*))^k $*_a$ *uint64-nat-assn*^{k} $*_a$ *nat-assn*^{k} → *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*)) ∈
      [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
      (*arlO-assn* (*array-assn R*))^k $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *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′ ≡ λ*a i x*.
   Array.nth′ a (*integer-of-uint32 i*) ⪢=
   (λ*j*. *arl-append j x* ⪢=
      (λ*a′*. *Array.upd′ a* (*integer-of-uint32 i*) *a′* ⪢= (λ-. *return a*)))

lemma *append-el-aa-append-el-aa-u′*:
  ‹*append-el-aa xs* (*nat-of-uint32 i*) *j* = *append-el-aa-u′ xs i j*›
  ⟨*proof*⟩

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

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

lemma *append-el-aa-u′-code*[*code*]:
  append-el-aa-u′ = (λ*a i x*. *nth-u-code a i* ⪢=
    (λ*j*. *arl-append j x* ⪢=
     (λ*a′*. *heap-array-set′-u a i a′* ⪢= (λ-. *return a*))))
  ⟨*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 $\gg$= *arl-set-u a i*
}›

lemma *update-raa-u32-rule*[*sep-heap-rules*]:
  assumes *p*: ‹*is-pure R*› **and** ‹*bb* < *length a*› **and** ‹*ba* < *length-rll a bb*› **and**
    ‹(*bb'*, *bb*) ∈ *uint32-nat-rel*›
  shows ‹<*R b bi* ∗ *arlO-assn* (*array-assn R*) *a ai*> *update-raa-u32 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-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*
    <λ*r*. *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*)}›

definition (**in** −)*length-uint32-nat* **where**
  [*simp*]: ‹*length-uint32-nat C* = *length C*›

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

definition *length-u* **where**
  [*simp*]: ‹*length-u xs* = *length xs*›

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

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

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*)
   ∈ [λ*C*. *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*)) ∈
    [λ(*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*)) ∈
    [λ(*xs*, *i*). *i* < *length xs* ∧ *length* (*xs* ! *i*) ≤ *uint32-max*]$_a$

$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn\rangle$
⟨proof⟩

TODO: proper fix to avoid the conversion to uint32

definition $length\text{-}aa\text{-}u\text{-}code :: \langle('a::heap\ array)\ array\text{-}list \Rightarrow nat \Rightarrow uint32\ Heap\rangle$ **where**
$\langle length\text{-}aa\text{-}u\text{-}code\ xs\ i = do\ \{$
$n \leftarrow length\text{-}raa\ xs\ i;$
$return\ (uint32\text{-}of\text{-}nat\ n)\}\rangle$

64-bits **definition** (**in** $-$)$length\text{-}aa\text{-}u64 :: \langle('a::heap\ array\text{-}list)\ array \Rightarrow uint64 \Rightarrow nat\ Heap\rangle$ **where**
$\langle length\text{-}aa\text{-}u64\ xs\ i = length\text{-}aa\ xs\ (nat\text{-}of\text{-}uint64\ i)\rangle$

lemma $length\text{-}aa\text{-}u64\text{-}code[code]$:
$\langle length\text{-}aa\text{-}u64\ xs\ i = nth\text{-}u64\text{-}code\ xs\ i \gg= arl\text{-}length\rangle$
⟨proof⟩

lemma $length\text{-}aa\text{-}u64\text{-}hnr[sepref\text{-}fr\text{-}rules]$: $\langle(uncurry\ length\text{-}aa\text{-}u64,\ uncurry\ (RETURN \circ\circ length\text{-}ll)) \in$
$[\lambda(xs,\ i).\ i < length\ xs]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow nat\text{-}assn\rangle$
⟨proof⟩

definition $length\text{-}raa\text{-}u64 :: \langle 'a::heap\ arrayO\text{-}raa \Rightarrow nat \Rightarrow uint64\ Heap\rangle$ **where**
$\langle length\text{-}raa\text{-}u64\ xs\ i = do\ \{$
$x \leftarrow arl\text{-}get\ xs\ i;$
$length\text{-}u64\text{-}code\ x\}\rangle$

lemma $length\text{-}raa\text{-}u64\text{-}alt\text{-}def$: $\langle length\text{-}raa\text{-}u64\ xs\ i = do\ \{$
$n \leftarrow length\text{-}raa\ xs\ i;$
$return\ (uint64\text{-}of\text{-}nat\ n)\}\rangle$
⟨proof⟩

definition $length\text{-}rll\text{-}n\text{-}uint64$ **where**
$[simp]$: $\langle length\text{-}rll\text{-}n\text{-}uint64 = length\text{-}rll\rangle$

lemma $length\text{-}raa\text{-}u64\text{-}hnr[sepref\text{-}fr\text{-}rules]$:
shows $\langle(uncurry\ length\text{-}raa\text{-}u64,\ uncurry\ (RETURN \circ\circ length\text{-}rll\text{-}n\text{-}uint64)) \in$
$[\lambda(xs,\ i).\ i < length\ xs \wedge length\ (xs\ !\ i) \le uint64\text{-}max]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn\rangle$
⟨proof⟩

Delete at index

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

lemma (**in** $-$) $delete\text{-}index\text{-}and\text{-}swap\text{-}alt\text{-}def$:
$\langle delete\text{-}index\text{-}and\text{-}swap\ S\ i =$
$(let\ x = last\ S\ in\ butlast\ (S[i := x]))\rangle$
⟨proof⟩

lemma $mset\text{-}tl\text{-}delete\text{-}index\text{-}and\text{-}swap$:
assumes
$\langle 0 < i\rangle$ **and**
$\langle i < length\ outl'\rangle$
shows $\langle mset\ (tl\ (delete\text{-}index\text{-}and\text{-}swap\ outl'\ i)) =$

remove1-mset (*outl'* ! *i*) (*mset* (*tl outl'*))›
⟨*proof*⟩

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

definition *swap-aa-u64* :: (*'a::*{*heap,default*}) *arrayO-raa ⇒ nat ⇒ uint64 ⇒ uint64 ⇒ 'a arrayO-raa Heap* **where**
  ‹*swap-aa-u64 xs k i j = do* {
    xi ← arl-get xs k;
    xj ← swap-u64-code xi i j;
    xs ← arl-set xs k xj;
    return xs
  }›

lemma *swap-aa-u64-hnr*[*sepref-fr-rules*]:

assumes ‹*is-pure R*›
shows ‹(*uncurry3 swap-aa-u64*, *uncurry3* (*RETURN oooo swap-ll*)) ∈
$[\lambda(((xs, k), i), j).\ k < length\ xs \wedge i < length\text{-}rll\ xs\ k \wedge j < length\text{-}rll\ xs\ k]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow$
$(arlO\text{-}assn\ (array\text{-}assn\ R))$›
⟨*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*)) ∈
$[\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ xs]_a$
$(arl\text{-}assn\ R)^d *_a uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow arl\text{-}assn\ R$›
⟨*proof*⟩

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

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

definition *nth-raa-i32-u64* :: ⟨'a::heap arrayO-raa ⇒ uint32 ⇒ uint64 ⇒ 'a Heap⟩ **where**
⟨*nth-raa-i32-u64 xs i j = do* {
$x \leftarrow arl\text{-}get\text{-}u\ xs\ i$;
$y \leftarrow nth\text{-}u64\text{-}code\ x\ j$;
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*)) ∈
$[\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to R\rangle$
⟨*proof*⟩

definition *nth-raa-i32-u32* :: ⟨'a::heap arrayO-raa ⇒ uint32 ⇒ uint32 ⇒ 'a Heap⟩ **where**
⟨*nth-raa-i32-u32 xs i j = do* {
$x \leftarrow arl\text{-}get\text{-}u\ xs\ i$;
$y \leftarrow nth\text{-}u\text{-}code\ x\ j$;
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*)) ∈
$[\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \to R\rangle$
⟨*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* {
$x \leftarrow nth\text{-}u\text{-}code\ xs\ i$;
$y \leftarrow arl\text{-}get\text{-}u\ x\ j$;
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*)) ∈
$[\lambda((x, L), L').\ L < length\ x \wedge L' < length\ (x\ !\ L)]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \to R\rangle$
⟨*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*)) ∈
$[\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›
⟨*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*)) ∈
$[\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

lemma *nth-aa-i64-u64-code*[*code*]:
‹*nth-aa-i64-u64 x L L'* = *nth-u64-code x L* ⤜ (λ*x*. *arl-get-u64 x L'* ⤜ *return*)›
⟨*proof*⟩

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

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

Length **definition** *length-raa-i64-u* :: ‹'*a*::*heap arrayO-raa* ⇒ *uint64* ⇒ *uint32 Heap*› **where**
‹*length-raa-i64-u xs i* = *do* {

$x \leftarrow$ *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*
<λ*r*. *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*
<λ*r*. *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*
<λ*r*. *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* >>= (λ*n*. *return* (*uint64-of-nat n*))›

definition *arl-length-o64* **where**
‹*arl-length-o64 x* = *do* {*n* ← *arl-length x*; *return* (*uint64-of-nat n*)}›

lemma *length-aa-u64-o64-code*[*code*]:
‹*length-aa-u64-o64 xs i* = *nth-u64-code xs i* ≫= *arl-length-o64*›
⟨*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* >>= (λ*n*. *return* (*uint64-of-nat n*))›

lemma *length-aa-u32-o64-code*[*code*]:
‹*length-aa-u32-o64 xs i* = *nth-u-code xs i* ≫= *arl-length-o64*›
⟨*proof*⟩

lemma *length-aa-u32-o64-hnr*[*sepref-fr-rules*]:
‹(*uncurry length-aa-u32-o64*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈
$[\lambda(xs, i).\ i < length\ xs \land length\ (xs\ !\ i) \leq uint64\text{-}max]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨*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*]:
‹$b < length\ xs \Longrightarrow (b', b) \in uint32\text{-}nat\text{-}rel \Longrightarrow$ <*arlO-assn* (*array-assn R*) *xs a*> *length-raa-u32 a b'*
<λ*r*. *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*)) ∈
$[\lambda(xs, i).\ i < length\ xs]_a\ (arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow nat\text{-}assn$›
⟨*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*)) ∈
$[\lambda(xs, i).\ i < length\ xs \land length\ (xs\ !\ i) \leq uint64\text{-}max]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨*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*)) ∈
$[\lambda(xs, i).\ i < length\ xs \land length\ (xs\ !\ i) \leq uint64\text{-}max]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨*proof*⟩

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-u*) $\in$ [$\lambda xs.$ *length xs* $\leq$ *uint32-max*]$_a$ (*arlO-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*) $\in$
[$\lambda xs.$ *length xs* $\leq$ *uint64-max*]$_a$ (*arl-assn R*)^k $\rightarrow$ *uint64-nat-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' ← *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 ← *last-aa-u64 xs i*;
xs ← *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* {
$x \leftarrow nth\text{-}u\text{-}code\ a\ i$;
$a' \leftarrow arl\text{-}set\text{-}u64\ x\ j\ y$;
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* {
$x \leftarrow last\text{-}aa\text{-}u\ xs\ i$;
$xs \leftarrow update\text{-}aa\text{-}u32\text{-}i64\ xs\ i\ j\ x$;
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* {
$xi \leftarrow arl\text{-}get\text{-}u\ xs\ k$;
$xj \leftarrow swap\text{-}u64\text{-}code\ xi\ i\ j$;
$xs \leftarrow arl\text{-}set\text{-}u\ xs\ k\ xj$;
return xs
}›

lemma *swap-aa-i32-u64-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 swap-aa-i32-u64*, *uncurry3* (*RETURN oooo swap-ll*)) ∈
$[\lambda(((xs, k), i), j).\ k < length\ xs \wedge i < length\text{-}rll\ xs\ k \wedge j < length\text{-}rll\ xs\ k]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^d *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow$
$(arlO\text{-}assn\ (array\text{-}assn\ R))$›
⟨*proof*⟩

Conversion from list of lists of *nat* to list of lists of *uint64*

definition *op-map* :: ('b ⇒ 'a::default) ⇒ 'a ⇒ 'b list ⇒ 'a list nres **where**
 ‹op-map R e xs = do {
  let zs = replicate (length xs) e;
 (-, zs) ← $WHILE_T^{\lambda(i,zs).\ i \le length\ xs \wedge take\ i\ zs = map\ R\ (take\ i\ xs) \wedge}$ 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›

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

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

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

sepref-definition *array-uint64-of-nat-code*
 is *array-uint64-of-nat*
 :: ‹$[\lambda xs.\ \forall a \in set\ xs.\ a \le uint64\text{-}max]_a$
   $(array\text{-}assn\ nat\text{-}assn)^k \rightarrow array\text{-}assn\ uint64\text{-}nat\text{-}assn$›
 ⟨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*⟩

end
theory *WB-More-Refinement-List*
imports *Refine-Imperative-HOL.IICF Weidenbach-Book-Base.WB-List-More*
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.

lemma *swap-nth-irrelevant*:
‹*k* ≠ *i* ⟹ *k* ≠ *j* ⟹ *swap xs i j* ! *k* = *xs* ! *k*›
⟨*proof*⟩

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

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

lemma *swap-nth-if*:

$\langle i < \textit{length xs} \Longrightarrow j < \textit{length xs} \Longrightarrow \textit{swap xs i j} ! k =$
$(\textit{if } k = i \textit{ then xs} ! j \textit{ else if } k = j \textit{ then xs} ! i \textit{ else xs} ! k)\rangle$
$\langle proof\rangle$

lemma *drop-swap-irrelevant*:
$\langle k > i \Longrightarrow k > j \Longrightarrow \textit{drop } k \ (\textit{swap outl}' \ j \ i) = \textit{drop } k \ \textit{outl}'\rangle$
$\langle proof\rangle$

lemma *take-swap-relevant*:
$\langle k > i \Longrightarrow k > j \Longrightarrow \textit{take } k \ (\textit{swap outl}' \ j \ i) = \textit{swap } (\textit{take } k \ \textit{outl}') \ i \ j\rangle$
$\langle proof\rangle$

lemma *tl-swap-relevant*:
$\langle i > 0 \Longrightarrow j > 0 \Longrightarrow \textit{tl } (\textit{swap outl}' \ j \ i) = \textit{swap } (\textit{tl outl}') \ (i - 1) \ (j - 1)\rangle$
$\langle proof\rangle$

lemma *swap-only-first-relevant*:
$\langle b \geq i \Longrightarrow a < \textit{length xs} \Longrightarrow \textit{take } i \ (\textit{swap xs a b}) = \textit{take } i \ (xs[a := xs ! b])\rangle$
$\langle proof\rangle$

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*:
$\langle [\![\textit{from} \leq \textit{length xs}; i < \textit{to} - \textit{from}]\!] \Longrightarrow \textit{Misc.slice from to xs} ! i = xs ! (\textit{from} + i)\rangle$
$\langle proof\rangle$

lemma *slice-irrelevant*[*simp*]:
$\langle i < \textit{from} \Longrightarrow \textit{Misc.slice from to } (xs[i := C]) = \textit{Misc.slice from to xs}\rangle$
$\langle i \geq \textit{to} \Longrightarrow \textit{Misc.slice from to } (xs[i := C]) = \textit{Misc.slice from to xs}\rangle$
$\langle i \geq \textit{to} \lor i < \textit{from} \Longrightarrow \textit{Misc.slice from to } (xs[i := C]) = \textit{Misc.slice from to xs}\rangle$
$\langle proof\rangle$

lemma *slice-update-swap*[*simp*]:
$\langle i < \textit{to} \Longrightarrow i \geq \textit{from} \Longrightarrow i < \textit{length xs} \Longrightarrow$
$\textit{Misc.slice from to } (xs[i := C]) = (\textit{Misc.slice from to xs})[(i - \textit{from}) := C]\rangle$
$\langle proof\rangle$

lemma *drop-slice*[*simp*]:
$\langle \textit{drop } n \ (\textit{Misc.slice from to xs}) = \textit{Misc.slice } (\textit{from} + n) \ \textit{to xs}\rangle$ **for** *from n to xs*
$\langle proof\rangle$

lemma *take-slice*[*simp*]:
$\langle \textit{take } n \ (\textit{Misc.slice from to xs}) = \textit{Misc.slice from } (\textit{min to } (\textit{from} + n)) \ xs\rangle$ **for** *from n to xs*
$\langle proof\rangle$

lemma *slice-append*[*simp*]:
$\langle \textit{to} \leq \textit{length xs} \Longrightarrow \textit{Misc.slice from to } (xs \ @ \ ys) = \textit{Misc.slice from to xs}\rangle$
$\langle proof\rangle$

lemma *slice-prepend*[*simp*]:
$\langle \textit{from} \geq \textit{length xs} \Longrightarrow$
$\textit{Misc.slice from to } (xs \ @ \ ys) = \textit{Misc.slice } (\textit{from} - \textit{length xs}) \ (\textit{to} - \textit{length xs}) \ ys\rangle$
$\langle proof\rangle$

lemma *slice-len-min-If*:
$\langle \textit{length } (\textit{Misc.slice from to xs}) =$

$(if\ from < length\ xs\ then\ min\ (length\ xs - from)\ (to - from)\ else\ 0)$›
⟨proof⟩

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

lemma *slice-end-length*: ‹$n \geq length\ xs \Longrightarrow Misc.slice\ to\ n\ xs = drop\ to\ xs$›
⟨proof⟩

lemma *slice-swap*[*simp*]:
‹$l \geq from \Longrightarrow l < to \Longrightarrow k \geq from \Longrightarrow k < to \Longrightarrow from < length\ arena \Longrightarrow$
$Misc.slice\ from\ to\ (swap\ arena\ l\ k) = swap\ (Misc.slice\ from\ to\ arena)\ (k - from)\ (l - from)$›
⟨proof⟩

lemma *drop-swap-relevant*[*simp*]:
‹$i \geq k \Longrightarrow j \geq k \Longrightarrow j < length\ outl' \Longrightarrow drop\ k\ (swap\ outl'\ j\ i) = swap\ (drop\ k\ outl')\ (j - k)\ (i - k)$›
⟨proof⟩

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

lemma *in-mset-rel-eq-f-iff*:
‹$(a, b) \in \langle\{(c, a).\ a = f\ c\}\rangle mset\text{-}rel \longleftrightarrow b = f\ \text{‘\#}\ a$›
⟨proof⟩

lemma *in-mset-rel-eq-f-iff-set*:
‹$\langle\{(c, a).\ a = f\ c\}\rangle mset\text{-}rel = \{(b, a).\ a = f\ \text{‘\#}\ b\}$›
⟨proof⟩

end
theory *Watched-Literals-Transition-System*
imports *Refine-Imperative-HOL.IICF 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* **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.

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**
‹$D \in\# N \Longrightarrow$ *update-clauses* $(N,\ U)$ D L L' (*add-mset* (*update-clause* D L L') (*remove1-mset* D N), U)›
| ‹$D \in\# U \Longrightarrow$ *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, $\{\#(L,\ C) | C \in\# N + U.\ L \in\#$ *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* $(D' - \{\#L\#\}) \equiv i$⟩ **and**
⟨*get-level M K* $= i + 1$⟩
⟨$D' \neq \{\#L\#\}$⟩ **and**
⟨$D' \subseteq\# D$⟩ **and**
⟨*clause* ‘# $(N + U) + NE + UE \models pm\ D'$⟩ **and**
⟨$L \in\# D'$⟩
⟨$L' \in\# D'$⟩ **and** — L' is the new watched literal
⟨*get-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 $W = 2 \wedge$ *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 $\longleftrightarrow$
  $(\exists L.\ L \in\# Q \wedge L \in\#$ *watched* $C) \vee (\exists L.\ (L, C) \in\# WS)$›

definition *is-blit* :: ‹$('a, 'b)$ *ann-lits* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool*›**where**
  [*simp*]: ‹*is-blit* M D L $\longleftrightarrow$ $(L \in\# D \wedge L \in$ *lits-of-l* $M)$›

definition *has-blit*:: ‹$('a, 'b)$ *ann-lits* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool*›**where**
  ‹*has-blit* M D L' $\longleftrightarrow$ $(\exists L.$ *is-blit* M D L $\wedge$ *get-level* M L $\leq$ *get-level* M $L')$›

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

fun *twl-lazy-update* :: ‹$('a, 'b)$ *ann-lits* $\Rightarrow$ $'a$ *twl-cls* $\Rightarrow$ *bool*› **where**
‹*twl-lazy-update* M (*TWL-Clause* W UW) $\longleftrightarrow$
  $(\forall L.\ L \in\# W \longrightarrow -L \in$ *lits-of-l* M $\longrightarrow \neg$*has-blit* M $(W+UW)$ L $\longrightarrow$
    $(\forall K \in\# UW.$ *get-level* M L $\geq$ *get-level* M K $\wedge$ $-K \in$ *lits-of-l* $M))$›

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

fun *watched-literals-false-of-max-level* :: ‹$('a, 'b)$ *ann-lits* $\Rightarrow$ $'a$ *twl-cls* $\Rightarrow$ *bool*› **where**
‹*watched-literals-false-of-max-level* M (*TWL-Clause* W UW) $\longleftrightarrow$
  $(\forall L.\ L \in\# W \longrightarrow -L \in$ *lits-of-l* M $\longrightarrow \neg$*has-blit* M $(W+UW)$ L $\longrightarrow$
    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* $\Rightarrow$ *bool*› **where**
‹*no-duplicate-queued* $(M, N, U, D, NE, UE, WS, Q) \longleftrightarrow$
  $(\forall C\ C'.\ C \in\# WS \longrightarrow C' \in\# WS \longrightarrow$ *fst* C $=$ *fst* $C') \wedge$
  $(\forall C.\ C \in\# WS \longrightarrow$ *add-mset* (*fst* C) $Q \subseteq\#$ *uminus* ‘# *lit-of* ‘# *mset* $M) \wedge$
  $Q \subseteq\#$ *uminus* ‘# *lit-of* ‘# *mset* M›

lemma *no-duplicate-queued-alt-def*:
  ‹*no-duplicate-queued* S $=$
   $((\forall C\ C'.\ C \in\#$ *clauses-to-update* $S \longrightarrow C' \in\#$ *clauses-to-update* $S \longrightarrow$ *fst* C $=$ *fst* $C') \wedge$
    $(\forall C.\ C \in\#$ *clauses-to-update* $S \longrightarrow$ *add-mset* (*fst* C) (*literals-to-update* S) $\subseteq\#$ *uminus* ‘# *lit-of* ‘# *mset* (*get-trail* S)) $\wedge$
    literals-to-update S $\subseteq\#$ *uminus* ‘# *lit-of* ‘# *mset* (*get-trail* S))›
  ⟨*proof*⟩

fun *distinct-queued* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*distinct-queued* $(M, N, U, D, NE, UE, WS, Q) \longleftrightarrow$
  distinct-mset Q $\wedge$
  $(\forall L\ C.$ *count* WS $(L, C) \leq$ *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**

‹$clauses\text{-}to\text{-}update\text{-}prop\ Q\ M\ (L,\ C) \longleftrightarrow$
$(L \in\#\ watched\ C \land -L \in lits\text{-}of\text{-}l\ M \land L \notin\#\ Q \land \neg has\text{-}blit\ M\ (clause\ C)\ L)$›
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* :: ‹$'v\ twl\text{-}st \Rightarrow bool$› **where**
‹$clauses\text{-}to\text{-}update\text{-}inv\ (M,\ N,\ U,\ None,\ NE,\ UE,\ WS,\ Q) \longleftrightarrow$
$(\forall L\ C.\ ((L,\ C) \in\#\ WS \longrightarrow \{\#(L,\ C)|\ C \in\#\ N + U.\ clauses\text{-}to\text{-}update\text{-}prop\ Q\ M\ (L,\ C)\#\} \subseteq\#\ WS)) \land$
$(\forall L.\ WS = \{\#\} \longrightarrow \{\#(L,\ C)|\ C \in\#\ N + U.\ clauses\text{-}to\text{-}update\text{-}prop\ Q\ M\ (L,\ C)\#\} = \{\#\}) \land$
$(\forall L\ C.\ C \in\#\ N + U \longrightarrow L \in\#\ watched\ C \longrightarrow -L \in lits\text{-}of\text{-}l\ M \longrightarrow \neg has\text{-}blit\ M\ (clause\ C)\ L \longrightarrow$
$(L,\ C) \notin\#\ WS \longrightarrow L \in\#\ Q)$›
| ‹$clauses\text{-}to\text{-}update\text{-}inv\ (M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \longleftrightarrow True$›

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

fun *twl-exception-inv* :: ‹$'v\ twl\text{-}st \Rightarrow\ 'v\ twl\text{-}cls \Rightarrow bool$› **where**
‹$twl\text{-}exception\text{-}inv\ (M,\ N,\ U,\ None,\ NE,\ UE,\ WS,\ Q)\ C \longleftrightarrow$
$(\forall L.\ L \in\#\ watched\ C \longrightarrow -L \in lits\text{-}of\text{-}l\ M \longrightarrow \neg has\text{-}blit\ M\ (clause\ C)\ L \longrightarrow$
$L \notin\#\ Q \longrightarrow (L,\ C) \notin\#\ WS \longrightarrow$
$(\forall K \in\#\ unwatched\ C.\ -K \in lits\text{-}of\text{-}l\ M))$›
| ‹$twl\text{-}exception\text{-}inv\ (M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q)\ C \longleftrightarrow True$›

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

fun *twl-st-exception-inv* :: ‹$'v\ twl\text{-}st \Rightarrow bool$› **where**
‹$twl\text{-}st\text{-}exception\text{-}inv\ (M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \longleftrightarrow$
$(\forall C \in\#\ N + U.\ twl\text{-}exception\text{-}inv\ (M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q)\ C)$›

Candidats for propagation (i.e., the clause where only one literals is non assigned) are enqueued.

fun *propa-cands-enqueued* :: ‹$'v\ twl\text{-}st \Rightarrow bool$› **where**
‹$propa\text{-}cands\text{-}enqueued\ (M,\ N,\ U,\ None,\ NE,\ UE,\ WS,\ Q) \longleftrightarrow$
$(\forall L\ C.\ C \in\#\ N{+}U \longrightarrow L \in\#\ clause\ C \longrightarrow M \models as\ CNot\ (remove1\text{-}mset\ L\ (clause\ C)) \longrightarrow$
$undefined\text{-}lit\ M\ L \longrightarrow$
$(\exists L'.\ L' \in\#\ watched\ C \land L' \in\#\ Q) \lor (\exists L.\ (L,\ C) \in\#\ WS))$›
| ‹$propa\text{-}cands\text{-}enqueued\ (M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \longleftrightarrow True$›

fun *confl-cands-enqueued* :: ‹$'v\ twl\text{-}st \Rightarrow bool$› **where**
‹$confl\text{-}cands\text{-}enqueued\ (M,\ N,\ U,\ None,\ NE,\ UE,\ WS,\ Q) \longleftrightarrow$
$(\forall C \in\#\ N + U.\ M \models as\ CNot\ (clause\ C) \longrightarrow$
$(\exists L'.\ L' \in\#\ watched\ C \land L' \in\#\ Q) \lor (\exists L.\ (L,\ C) \in\#\ WS))$›
| ‹$confl\text{-}cands\text{-}enqueued\ (M,\ N,\ U,\ Some\ \text{-},\ NE,\ UE,\ WS,\ Q) \longleftrightarrow$
$True$›

This invariant talk about the decomposition of the trail and the invariants that holds in these states.

fun *past-invs* :: ⟨'*v twl-st* ⇒ *bool*⟩ **where**
  ⟨*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⟷
    (∀ *M1 M2 K*. *M* = *M2* @ *Decided K* # *M1* ⟶ (
      (∀ *C* ∈# *N* + *U*. *twl-lazy-update M1 C* ∧
        watched-literals-false-of-max-level M1 C ∧
        twl-exception-inv (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*) ∧
      confl-cands-enqueued (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) ∧
      propa-cands-enqueued (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) ∧
      clauses-to-update-inv (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})))⟩
declare *past-invs.simps*[*simp del*]

fun *twl-st-inv* :: ⟨'*v twl-st* ⇒ *bool*⟩ **where**
⟨*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⟷
  (∀ *C* ∈# *N* + *U*. *struct-wf-twl-cls C*) ∧
  (∀ *C* ∈# *N* + *U*. *D* = *None* ⟶ ¬*twl-is-an-exception C Q WS* ⟶ (*twl-lazy-update M C*)) ∧
  (∀ *C* ∈# *N* + *U*. *D* = *None* ⟶ *watched-literals-false-of-max-level M C*)⟩

lemma *twl-st-inv-alt-def*:
  ⟨*twl-st-inv S* ⟷
  (∀ *C* ∈# *get-clauses S*. *struct-wf-twl-cls C*) ∧
  (∀ *C* ∈# *get-clauses S*. *get-conflict S* = *None* ⟶
    ¬*twl-is-an-exception C* (*literals-to-update S*) (*clauses-to-update S*) ⟶
    (*twl-lazy-update* (*get-trail S*) *C*)) ∧
  (∀ *C* ∈# *get-clauses S*. *get-conflict S* = *None* ⟶
    watched-literals-false-of-max-level (*get-trail S*) *C*)⟩
  ⟨*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) $\longleftrightarrow$ *distinct-mset* C›
⟨*proof*⟩

lemma *distinct-image-mset-clause*:
‹*distinct-mset* (*clause* ‘# C) $\Longrightarrow$ *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 $\Longrightarrow$ *twl-struct-invs* S $\Longrightarrow$
cdcl$_W$-restart-mset.cdcl$_W$-stgy (*state$_W$-of* S) (*state$_W$-of* T) $\vee$
(*state$_W$-of* S = *state$_W$-of* T $\wedge$ (*literals-to-update-measure* T, *literals-to-update-measure* S)
$\in$ *lexn less-than 2*)›
⟨*proof*⟩

lemma *cdcl-twl-cp-conflict*:
‹*cdcl-twl-cp* S T $\Longrightarrow$ *get-conflict* $T \neq$ *None* $\longrightarrow$
clauses-to-update T = {#} $\wedge$ *literals-to-update* T = {#}›
⟨*proof*⟩

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

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

lemma *cdcl-twl-cp-twl-struct-invs*:
‹*cdcl-twl-cp* S T $\Longrightarrow$ *twl-struct-invs* S $\Longrightarrow$ *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 $\Longrightarrow$ *twl-struct-invs* S $\Longrightarrow$ *twl-stgy-invs* S $\Longrightarrow$ *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* $T \neq$ *None* $\Longrightarrow$ *clauses-to-update* T = {#} $\wedge$ *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-twl-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-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 *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* $\{(T,\ S{::}'v\ \textit{twl-st}).\ \textit{twl-struct-invs}\ S \wedge \textit{cdcl-twl-o}\ S\ T\}$⟩
⟨*proof*⟩

lemma *tranclp-wf-cdcl-twl-o*:
⟨*wf* $\{(T,\ S{::}'v\ \textit{twl-st}).\ \textit{twl-struct-invs}\ S \wedge \textit{cdcl-twl-o}^{++}\ S\ T\}$⟩
⟨*proof*⟩

lemma (**in** −)*propa-cands-enqueued-mono*:
⟨$U' \subseteq\#\ U \Longrightarrow N' \subseteq\#\ N \Longrightarrow$
propa-cands-enqueued $(M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \Longrightarrow$
propa-cands-enqueued $(M,\ N',\ U',\ D,\ NE',\ UE',\ WS,\ Q)$⟩
⟨*proof*⟩

lemma (**in** −)*confl-cands-enqueued-mono*:
⟨$U' \subseteq\#\ U \Longrightarrow N' \subseteq\#\ N \Longrightarrow$
confl-cands-enqueued $(M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \Longrightarrow$
confl-cands-enqueued $(M,\ N',\ U',\ D,\ NE',\ UE',\ WS,\ Q)$⟩
⟨*proof*⟩

lemma (**in** −)*twl-st-exception-inv-mono*:
⟨$U' \subseteq\#\ U \Longrightarrow N' \subseteq\#\ N \Longrightarrow$
twl-st-exception-inv $(M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \Longrightarrow$
twl-st-exception-inv $(M,\ N',\ U',\ D,\ NE',\ UE',\ WS,\ Q)$⟩
⟨*proof*⟩

lemma (**in** −)*twl-st-inv-mono*:
⟨$U' \subseteq\#\ U \Longrightarrow N' \subseteq\#\ N \Longrightarrow$
twl-st-inv $(M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q) \Longrightarrow$
twl-st-inv $(M,\ N',\ U',\ D,\ NE',\ UE',\ WS,\ Q)$⟩
⟨*proof*⟩

lemma (**in** −) *rtranclp-cdcl-twl-stgy-twl-stgy-invs*:
assumes
⟨$\textit{cdcl-twl-stgy}^{**}\ S\ T$⟩ **and**
⟨*twl-struct-invs* S⟩ **and**
⟨*twl-stgy-invs* S⟩
shows ⟨*twl-stgy-invs* T⟩
⟨*proof*⟩

lemma *after-fast-restart-replay*:
assumes
inv: ⟨$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-all-struct-inv}\ (M',\ N,\ U,\ None)$⟩ **and**
stgy-invs: ⟨$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-stgy-invariant}\ (M',\ N,\ U,\ None)$⟩ **and**
smaller-propa: ⟨$\textit{cdcl}_W\textit{-restart-mset.no-smaller-propa}\ (M',\ N,\ U,\ None)$⟩ **and**
kept: ⟨$\forall L\ E.\ Propagated\ L\ E \in set\ (drop\ (length\ M' - n)\ M') \longrightarrow E \in\#\ N + U'$⟩ **and**
U'-U: ⟨$U' \subseteq\#\ U$⟩
shows
⟨$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-stgy}^{**}\ ([],\ N,\ U',\ None)\ (drop\ (length\ M' - n)\ M',\ N,\ U',\ None)$⟩
⟨*proof*⟩

lemma *after-fast-restart-replay-no-stgy*:
assumes
inv: ⟨$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-all-struct-inv}\ (M',\ N,\ U,\ None)$⟩ **and**
kept: ⟨$\forall L\ E.\ Propagated\ L\ E \in set\ (drop\ (length\ M' - n)\ M') \longrightarrow E \in\#\ N + U'$⟩ **and**
U'-U: ⟨$U' \subseteq\#\ U$⟩
shows

$\langle cdcl_W\text{-}restart\text{-}mset.cdcl_W^{**}$ ([], N, U', $None$) ($drop$ ($length$ $M' - n$) M', N, U', $None$)$\rangle$
⟨*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 ⟨$cdcl\text{-}twl\text{-}stgy^{**}$ *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 ⟨$cdcl\text{-}twl\text{-}stgy^{**}$ *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 ⟨$cdcl\text{-}twl\text{-}stgy^{**}$ *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 $cdcl\text{-}twl\text{-}stgy\text{-}cdcl_W\text{-}stgy3$:
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* $S \longleftrightarrow$
no-step cdcl-twl-stgy $S \lor$ (*get-conflict* $S \neq$ *None* $\land$ *count-decided* (*get-trail* S) = *0*)⟩

definition *conclusive-TWL-run* :: ⟨$'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st nres*⟩ **where**
⟨*conclusive-TWL-run* S = *SPEC*(λT. *cdcl-twl-stgy*** S T $\land$ *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* $S \neq$ *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* $S \neq$ *None*⟩
shows ⟨*unsatisfiable* (*set-mset* (*init-clss* S + *learned-clss* S))⟩
⟨*proof*⟩

end
theory *Watched-Literals-Algorithm*
imports
Watched-Literals-Transition-System
WB-More-Refinement
begin

1.2 First Refinement: Deterministic Rule Application

1.2.1 Unit Propagation Loops

definition *set-conflicting* :: ⟨$'v$ *twl-cls* $\Rightarrow$ $'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st*⟩ **where**
⟨*set-conflicting* = (λC (M, N, U, D, NE, UE, WS, Q). (M, N, U, *Some* (*clause* C), NE, UE, $\{\#\}$, $\{\#\}$))⟩

definition *propagate-lit* :: ⟨$'v$ *literal* $\Rightarrow$ $'v$ *twl-cls* $\Rightarrow$ $'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st*⟩ **where**
⟨*propagate-lit* = ($\lambda L'$ 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.
($L \neq$ *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* S_0 =
do {
(-, *S*) ←
$WHILE_T{}^{skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\ S_0}$
(λ(*uip*, *S*). ¬*uip* ∧ ¬*is-decided* (*hd* (*get-trail S*)))
(λ(-, *S*).
do {
ASSERT(*get-trail S* ≠ []);
let D′ = *the* (*get-conflict S*);
(*L*, *C*) ← *SPEC*(λ(*L*, *C*). *Propagated L C* = *hd* (*get-trail S*));
if −*L* ∉# *D*′ *then*
do {*RETURN* (*False*, *tl-state S*)}
else
if get-maximum-level (*get-trail S*) (*remove1-mset* (−*L*) *D*′) = *count-decided* (*get-trail S*)
then
do {*RETURN* (*False*, *update-confl-tl* (*Some* ($cdcl_W$*-restart-mset.resolve-cls L D*′ *C*)) *S*)}
else
do {*RETURN* (*True*, *S*)}
}
)
(*False*, S_0);
RETURN S
}
⟩

lemma *skip-and-resolve-loop-spec*:
assumes *struct-S*: ⟨*twl-struct-invs S*⟩ **and** *stgy-S*: ⟨*twl-stgy-invs S*⟩ **and**
⟨*clauses-to-update S* = {#}⟩ **and** ⟨*literals-to-update S* = {#}⟩ **and**
⟨*get-conflict S* ≠ *None*⟩ **and** *count-dec*: ⟨*count-decided* (*get-trail S*) > *0*⟩
shows ⟨*skip-and-resolve-loop S* ≤ *SPEC*(λ*T*. *cdcl-twl-o*** *S T* ∧ *twl-struct-invs T* ∧ *twl-stgy-invs T*
∧
no-step $cdcl_W$*-restart-mset.skip* ($state_W$*-of T*) ∧
no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of T*) ∧
get-conflict T ≠ *None* ∧ *clauses-to-update T* = {#} ∧ *literals-to-update T* = {#})⟩
⟨*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 {
    $M1 \leftarrow SPEC(\lambda M1. \exists K\ M2.$ (Decided K # $M1$, $M2$) $\in$ set (get-all-ann-decomposition M) $\wedge$
      get-level M K = get-maximum-level M (the D' − {#−L#}) + 1);
    RETURN ($M1$, N, U, D', NE, UE, WS, Q)
})›

definition *propagate-bt* :: ‹'v literal ⇒ '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#} ∧ $L \neq -L'$ ∧
        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*) ∧

$(\neg\ brk \longrightarrow$ *get-conflict* $T = None \wedge (\forall S'.\ \neg$ *cdcl-twl-o* $T\ S')) \wedge$
$(brk \longrightarrow$ *get-conflict* $T \neq None \vee (\forall S'.\ \neg$ *cdcl-twl-stgy* $T\ S')) \wedge$
twl-struct-invs $T \wedge$ *twl-stgy-invs* $T \wedge$ *clauses-to-update* $T = \{\#\} \wedge$
$(\neg\ brk \longrightarrow$ *literals-to-update* $T \neq \{\#\}) \wedge$
$(\neg brk \longrightarrow \neg\ (\forall S'.\ \neg$ *cdcl-twl-o* $S\ S') \longrightarrow$ *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* $S \leq SPEC($*cdcl-twl-o-prog-spec* $S)$›
(**is** ‹- $\leq$ *?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* $S_0 \equiv \lambda(brk,\ T).$ *twl-struct-invs* $T \wedge$ *twl-stgy-invs* $T \wedge$
$(brk \longrightarrow$ *final-twl-state* $T) \wedge$ *cdcl-twl-stgy*** $S_0\ T \wedge$ *clauses-to-update* $T = \{\#\} \wedge$
$(\neg brk \longrightarrow$ *get-conflict* $T = None)$›

definition *cdcl-twl-stgy-prog* :: ‹*'v twl-st* $\Rightarrow$ *'v twl-st nres*› **where**
‹*cdcl-twl-stgy-prog* S_0 =
do {
  do {
    $(brk,\ T) \leftarrow WHILE_T{}^{cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}inv\ S_0}$
      $(\lambda(brk,\ \text{-}).\ \neg brk)$
      $(\lambda(brk,\ S).$
      do {
        $T \leftarrow$ *unit-propagation-outer-loop* S;
        cdcl-twl-o-prog T
      })
      $(False,\ S_0)$;
    RETURN T
  }
}
›

lemma *wf-cdcl-twl-stgy-measure*:
‹*wf* $(\{((brkT,\ T),\ (brkS,\ S)).$ *twl-struct-invs* $S \wedge$ *cdcl-twl-stgy*$^{++}$ $S\ T\}$
$\cup \{((brkT,\ T),\ (brkS,\ S)).\ S = T \wedge brkT \wedge \neg brkS\})$›
(**is** ‹*wf* (*?TWL* $\cup$ *?BOOL*)›)
⟨*proof*⟩

lemma *cdcl-twl-o-final-twl-state*:
assumes
‹*cdcl-twl-stgy-prog-inv* S $(brk,\ T)$› **and**
‹*case* $(brk,\ T)$ *of* $(brk,\ \text{-}) \Rightarrow \neg\ brk$› **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-List*
imports *Watched-Literals-Algorithm CDCL.DPLL-CDCL-W-Implementation*
begin

lemma *mset-take-mset-drop-mset*: ‹(λx. *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′* = (λ*C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*). (*M*, *N*, *U*, *C*, *NE*, *UE*, *WS*, *Q*))›

abbreviation *watched-l* :: ‹′*a clause-l* ⇒ ′*a clause-l*› **where**
‹*watched-l l* ≡ *take 2 l*›

abbreviation *unwatched-l* :: ‹′*a clause-l* ⇒ ′*a clause-l*› **where**
‹*unwatched-l l* ≡ *drop 2 l*›

fun *twl-clause-of* :: ‹′*a clause-l* ⇒ ′*a clause twl-clause*› **where**
‹*twl-clause-of l* = *TWL-Clause* (*mset* (*watched-l l*)) (*mset* (*unwatched-l l*))›

fun *clause-of* :: ‹′*a*::*plus twl-clause* ⇒ ′*a*› **where**
‹*clause-of* (*TWL-Clause W UW*) = *W* + *UW*›

abbreviation *clause-in* :: ‹′*v clauses-l* ⇒ *nat* ⇒ ′*v clause-l*› (**infix** ∝ *101*) **where**
‹$N \propto i \equiv$ *fst* (*the* (*fmlookup* N i))›

abbreviation *clause-upd* :: ‹′*v clauses-l* ⇒ *nat* ⇒ ′*v clause-l* ⇒ ′*v clauses-l*› **where**
‹*clause-upd* N i $C \equiv$ *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* ($N \propto C$)› **and** ‹$C \neq 0$› |
‹*convert-lit* N E (*Propagated* K C) (*Propagated* K C')›
if ‹$C = 0$› **and** ‹$C' \in\#\ E$›

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′* ⟹

$(M1\ @\ M2,\ M1'\ @\ M2') \in$ *convert-lits-l N E* $\longleftrightarrow (M1,\ M1') \in$ *convert-lits-l N E* $\wedge$
$(M2,\ M2') \in$ *convert-lits-l N E* ›
⟨*proof*⟩

lemma *convert-lits-l-map-lit-of*: ‹$(ay,\ bq) \in$ *convert-lits-l N e* $\Longrightarrow$ *map lit-of ay* = *map lit-of bq*›
⟨*proof*⟩

lemma *convert-lits-l-tlD*:
‹$(M,\ M') \in$ *convert-lits-l N E* $\Longrightarrow$
(*tl M*, *tl M'*) $\in$ *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* $\equiv$ *union-mset-list* (*remove1* $(-L)$ *D'*) (*remove1 L E*)›

lemma *mset-resolve-cls-l-resolve-cls*[*iff*]:
‹*mset* (*resolve-cls-l L D' E*) = $cdcl_W$*-restart-mset.resolve-cls L* (*mset D'*) (*mset E*)›
⟨*proof*⟩

lemma *resolve-cls-l-nil-iff*:
‹*resolve-cls-l L D' E* = [] $\longleftrightarrow$ $cdcl_W$*-restart-mset.resolve-cls L* (*mset D'*) (*mset E*) = {#}›
⟨*proof*⟩

lemma *lit-of-convert-lit*[*simp*]:
‹*convert-lit N E L L'* $\Longrightarrow$ *lit-of L'* = *lit-of L*›
⟨*proof*⟩

lemma *is-decided-convert-lit*[*simp*]:
‹*convert-lit N E L L'* $\Longrightarrow$ *is-decided L'* $\longleftrightarrow$ *is-decided L*›
⟨*proof*⟩

lemma *defined-lit-convert-lits-l*[*simp*]: ‹$(M,\ M') \in$ *convert-lits-l N E* $\Longrightarrow$
defined-lit M' = *defined-lit M*›
⟨*proof*⟩

lemma *no-dup-convert-lits-l*[*simp*]: ‹$(M,\ M') \in$ *convert-lits-l N E* $\Longrightarrow$
no-dup M' $\longleftrightarrow$ *no-dup M*›
⟨*proof*⟩

lemma
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*›
shows
count-decided-convert-lits-l[*simp*]:

$\langle$*count-decided* M' = *count-decided* $M\rangle$
$\langle$*proof*$\rangle$

lemma
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$
shows
get-level-convert-lits-l[*simp*]:
$\langle$*get-level* M' = *get-level* $M\rangle$
$\langle$*proof*$\rangle$

lemma
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$
shows
get-maximum-level-convert-lits-l[*simp*]:
$\langle$*get-maximum-level* M' = *get-maximum-level* $M\rangle$
$\langle$*proof*$\rangle$

lemma *list-of-l-convert-lits-l*[*simp*]:
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$
shows
$\langle$*lits-of-l* M' = *lits-of-l* $M\rangle$
$\langle$*proof*$\rangle$

lemma *is-proped-hd-convert-lits-l*[*simp*]:
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$ **and** $\langle M \neq []\rangle$
shows $\langle$*is-proped* (*hd* M') $\longleftrightarrow$ *is-proped* (*hd* M)$\rangle$
$\langle$*proof*$\rangle$

lemma *is-decided-hd-convert-lits-l*[*simp*]:
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$ **and** $\langle M \neq []\rangle$
shows
$\langle$*is-decided* (*hd* M') $\longleftrightarrow$ *is-decided* (*hd* M)$\rangle$
$\langle$*proof*$\rangle$

lemma *lit-of-hd-convert-lits-l*[*simp*]:
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$ **and** $\langle M \neq []\rangle$
shows
$\langle$*lit-of* (*hd* M') = *lit-of* (*hd* M)$\rangle$
$\langle$*proof*$\rangle$

lemma *lit-of-l-convert-lits-l*[*simp*]:
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$
shows
$\langle$*lit-of* ‘ *set* M' = *lit-of* ‘ *set* $M\rangle$
$\langle$*proof*$\rangle$

The order of the assumption is important for simpler use.

lemma *convert-lits-l-extend-mono*:
assumes $\langle(a,b) \in$ *convert-lits-l* N $E\rangle$
$\langle\forall L\ i.$ *Propagated* $L\ i \in$ *set* $a \longrightarrow$ *mset* $(N{\propto}i)$ = *mset* $(N'{\propto}i)\rangle$ **and** $\langle E \subseteq\#\ E'\rangle$
shows
$\langle(a,b) \in$ *convert-lits-l* N' $E'\rangle$
$\langle$*proof*$\rangle$

lemma *convert-lits-l-nil-iff*[*simp*]:
assumes $\langle(M,\ M') \in$ *convert-lits-l* N $E\rangle$

shows
‹$M' = [] \longleftrightarrow M = []$›
⟨*proof*⟩

lemma *convert-lits-l-atm-lits-of-l*:
assumes ‹$(M, M') \in$ *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*]:
‹$(M, M') \in$ *convert-lits-l N E* $\Longrightarrow M' \models as\ C \longleftrightarrow M \models as\ C$›
⟨*proof*⟩

lemma *convert-lit-propagated-decided*[*iff*]:
‹*convert-lit b d* (*Propagated x21 x22*) (*Decided x1*) $\longleftrightarrow$ *False*›
⟨*proof*⟩

lemma *convert-lit-decided*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Decided x2*) $\longleftrightarrow$ *x1* = *x2*›
⟨*proof*⟩

lemma *convert-lit-decided-propagated*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Propagated x21 x22*) $\longleftrightarrow$ *False*›
⟨*proof*⟩

lemma *convert-lits-l-lit-of-mset*[*simp*]:
‹$(a, af) \in$ *convert-lits-l N E* $\Longrightarrow$ *lit-of* ‘# *mset af* = *lit-of* ‘# *mset a*›
⟨*proof*⟩

lemma *convert-lits-l-imp-same-length*:
‹$(a, b) \in$ *convert-lits-l N E* $\Longrightarrow$ *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: ‹$(x, xa) \in$ *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* $\Longrightarrow$
(*M*, *TM*) ∈ *convert-lits-l N NE* $\Longrightarrow$
∃ *K'*. *K'* ∈ *set M* ∧ *convert-lit N NE K' K*›
⟨*proof*⟩

lemma *in-convert-lits-lD2*:
‹*K* ∈ *set M* $\Longrightarrow$
(*M*, *TM*) ∈ *convert-lits-l N NE* $\Longrightarrow$
∃ *K'*. *K'* ∈ *set TM* ∧ *convert-lit N NE K K'*›
⟨*proof*⟩

lemma *convert-lits-l-filter-decided*: ‹$(S, S') \in$ *convert-lits-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: ‹$C \in\#\ dom\text{-}m\ N$›
shows ‹$ran\text{-}m\ (N(C \hookrightarrow C')) = add\text{-}mset\ (C', irred\ N\ C)\ (remove1\text{-}mset\ (N \propto C, irred\ N\ C)\ (ran\text{-}m\ N))$›
⟨proof⟩

lemma *ran-m-mapsto-upd*:
assumes
NC: ‹$C \in\# \textit{ dom-m } N$›
shows ‹$\textit{ran-m } (\textit{fmupd } C\ C'\ N) =$
$\textit{add-mset } C'\ (\textit{remove1-mset } (N \propto C,\ \textit{irred } N\ C)\ (\textit{ran-m } N))$›
⟨*proof*⟩

lemma *ran-m-mapsto-upd-notin*:
assumes
NC: ‹$C \notin\# \textit{ dom-m } N$›
shows ‹$\textit{ran-m } (\textit{fmupd } C\ C'\ N) = \textit{add-mset } C'\ (\textit{ran-m } N)$›
⟨*proof*⟩

lemma *learned-clss-l-update*[*simp*]:
‹$bh \in\# \textit{ dom-m } ax \Longrightarrow \textit{size } (\textit{learned-clss-l } (ax(bh \hookrightarrow C))) = \textit{size } (\textit{learned-clss-l } ax)$›
⟨*proof*⟩

lemma *Ball-ran-m-dom*:
‹$(\forall x{\in}\#\textit{ran-m } N.\ P\ (\textit{fst } x)) \longleftrightarrow (\forall x{\in}\#\textit{dom-m } N.\ P\ (N \propto x))$›
⟨*proof*⟩

lemma *Ball-ran-m-dom-struct-wf*:
‹$(\forall x{\in}\#\textit{ran-m } N.\ \textit{struct-wf-twl-cls } (\textit{twl-clause-of } (\textit{fst } x))) \longleftrightarrow$
$(\forall x{\in}\#\textit{ dom-m } N.\ \textit{struct-wf-twl-cls } (\textit{twl-clause-of } (N \propto x)))$›
⟨*proof*⟩

lemma *init-clss-lf-fmdrop*[*simp*]:
‹$\textit{irred } N\ C \Longrightarrow C \in\# \textit{ dom-m } N \Longrightarrow \textit{init-clss-lf } (\textit{fmdrop } C\ N) = \textit{remove1-mset } (N{\propto}C)\ (\textit{init-clss-lf } N)$›
⟨*proof*⟩

lemma *init-clss-lf-fmdrop-irrelev*[*simp*]:
‹$\neg\textit{irred } N\ C \Longrightarrow \textit{init-clss-lf } (\textit{fmdrop } C\ N) = \textit{init-clss-lf } N$›
⟨*proof*⟩

lemma *learned-clss-lf-lf-fmdrop*[*simp*]:
‹$\neg\textit{irred } N\ C \Longrightarrow C \in\# \textit{ dom-m } N \Longrightarrow \textit{learned-clss-lf } (\textit{fmdrop } C\ N) = \textit{remove1-mset } (N{\propto}C)\ (\textit{learned-clss-lf } N)$›
⟨*proof*⟩

lemma *learned-clss-l-l-fmdrop*: ‹$\neg\ \textit{irred } N\ C \Longrightarrow C \in\# \textit{ dom-m } N \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmdrop } C\ N) = \textit{remove1-mset } (\textit{the } (\textit{fmlookup } N\ C))\ (\textit{learned-clss-l } N)$›
⟨*proof*⟩

lemma *learned-clss-lf-lf-fmdrop-irrelev*[*simp*]:
‹$\textit{irred } N\ C \Longrightarrow \textit{learned-clss-lf } (\textit{fmdrop } C\ N) = \textit{learned-clss-lf } N$›
⟨*proof*⟩

lemma *ran-mf-lf-fmdrop*[*simp*]:
‹$C \in\# \textit{ dom-m } N \Longrightarrow \textit{ ran-mf } (\textit{fmdrop } C\ N) = \textit{remove1-mset } (N{\propto}C)\ (\textit{ran-mf } N)$›
⟨*proof*⟩

lemma *ran-mf-lf-fmdrop-notin*[*simp*]:
‹$C \notin\# \textit{ dom-m } N \Longrightarrow \textit{ ran-mf } (\textit{fmdrop } C\ N) = \textit{ran-mf } N$›
⟨*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*⟩

definition *twl-st-l* :: ‹$- \Rightarrow ('v\ twl\text{-}st\text{-}l \times 'v\ twl\text{-}st)\ set$› **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 ‹$(S,\ T) \in twl\text{-}st\text{-}l\ L$›
shows
‹$(get\text{-}trail\text{-}l\ S,\ get\text{-}trail\ T) \in convert\text{-}lits\text{-}l\ (get\text{-}clauses\text{-}l\ S)\ (get\text{-}unit\text{-}clauses\text{-}l\ S)$› **and**
‹$get\text{-}clauses\ T = twl\text{-}clause\text{-}of\ `\#\ fst\ `\#\ ran\text{-}m\ (get\text{-}clauses\text{-}l\ S)$› **and**
‹$get\text{-}conflict\ T = get\text{-}conflict\text{-}l\ S$› **and**
‹$L = None \Longrightarrow clauses\text{-}to\text{-}update\ T = \{\#\}$›
‹$L \neq None \Longrightarrow clauses\text{-}to\text{-}update\ T =$

$(\lambda j.\ (the\ L,\ twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto j)))$ '# *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*)∈*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*) ∈ *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'*) ∈ *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*) ∈ *twl-st-l L* ⟹ *unit-init-clauses T* = *get-unit-init-clauses-l S*›
⟨*proof*⟩

lemma *clauses-state-to-l*[*twl-st-l*]: ‹(*S*, *S'*) ∈ *twl-st-l L* ⟹
cdcl$_W$*-restart-mset.clauses* (*state*$_W$*-of S'*) = *mset* '# *ran-mf* (*get-clauses-l S*) +
get-unit-init-clauses-l S + *get-unit-learned-clauses-l S*›
⟨*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*) ∈ *twl-st-l L* ⟹ *get-trail-l S* ≠ [] ⟹
lit-of (*hd* (*get-trail T*)) = *lit-of* (*hd* (*get-trail-l S*))›
⟨*proof*⟩

lemma *twl-st-l-mark-of-hd*:
‹(*x*, *y*) ∈ *twl-st-l b* ⟹
get-trail-l x ≠ [] ⟹
is-proped (*hd* (*get-trail-l x*)) ⟹
mark-of (*hd* (*get-trail-l x*)) > *0* ⟹
mark-of (*hd* (*get-trail y*)) = *mset* (*get-clauses-l x* ∝ *mark-of* (*hd* (*get-trail-l x*)))›
⟨*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* ⇒ ′*v twl-st-l* ⇒ *bool*› **where**
‹*equality-except-trail* (M, N, D, NE, UE, WS, Q) $(M', N', D', NE', UE', WS', Q')$ ⟷
$N = N' \land D = D' \land NE = NE' \land UE = UE' \land WS = WS' \land Q = Q'$›

fun *equality-except-conflict-l* :: ‹′*v twl-st-l* ⇒ ′*v twl-st-l* ⇒ *bool*› **where**
‹*equality-except-conflict-l* (M, N, D, NE, UE, WS, Q) $(M', N', D', NE', UE', WS', Q')$ ⟷
$M = M' \land N = N' \land NE = NE' \land UE = UE' \land WS = WS' \land Q = Q'$›

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 ⟷
get-trail-l S = *get-trail-l* T ∧ *get-clauses-l* S = *get-clauses-l* T ∧
get-unit-init-clauses-l S = *get-unit-init-clauses-l* T ∧
get-unit-learned-clauses-l S = *get-unit-learned-clauses-l* T ∧
literals-to-update-l S = *literals-to-update-l* T ∧
clauses-to-update-l S = *clauses-to-update-l* T›
⟨*proof*⟩

lemma *equality-except-conflict-alt-def*:
‹*equality-except-conflict* S T ⟷
get-trail S = *get-trail* T ∧ *get-init-clauses* S = *get-init-clauses* T ∧
get-learned-clss S = *get-learned-clss* T ∧
get-init-learned-clss S = *get-init-learned-clss* T ∧
unit-init-clauses S = *unit-init-clauses* T ∧
literals-to-update S = *literals-to-update* T ∧
clauses-to-update S = *clauses-to-update* T›
⟨*proof*⟩

1.3.2 Additional Invariants and Definitions

definition *twl-list-invs* **where**
‹*twl-list-invs* S ⟷

$(\forall C \in\# \ \textit{clauses-to-update-l}\ S.\ C \in\# \ \textit{dom-m}\ (\textit{get-clauses-l}\ S)) \wedge$
$0 \notin\# \ \textit{dom-m}\ (\textit{get-clauses-l}\ S) \wedge$
$(\forall L\ C.\ \textit{Propagated}\ L\ C \in \textit{set}\ (\textit{get-trail-l}\ S) \longrightarrow (C > 0 \longrightarrow C \in\# \ \textit{dom-m}\ (\textit{get-clauses-l}\ S) \wedge$
$(C > 0 \longrightarrow L \in \textit{set}\ (\textit{watched-l}\ (\textit{get-clauses-l}\ S \propto C)) \wedge L = \textit{get-clauses-l}\ S \propto C\ !\ 0))) \wedge$
$\textit{distinct-mset}\ (\textit{clauses-to-update-l}\ S)\rangle$

definition *polarity* **where**
$\langle \textit{polarity}\ M\ L =$
$(\textit{if undefined-lit}\ M\ L\ \textit{then None else if}\ L \in \textit{lits-of-l}\ M\ \textit{then Some True else Some False})\rangle$

lemma *polarity-None-undefined-lit*: $\langle \textit{is-None}\ (\textit{polarity}\ M\ L) \Longrightarrow \textit{undefined-lit}\ M\ L\rangle$
$\langle \textit{proof}\rangle$

lemma *polarity-spec*:
assumes $\langle \textit{no-dup}\ M\rangle$
shows
$\langle \textit{RETURN}\ (\textit{polarity}\ M\ L) \le \textit{SPEC}(\lambda v.\ (v = \textit{None} \longleftrightarrow \textit{undefined-lit}\ M\ L) \wedge$
$(v = \textit{Some True} \longleftrightarrow L \in \textit{lits-of-l}\ M) \wedge (v = \textit{Some False} \longleftrightarrow -L \in \textit{lits-of-l}\ M))\rangle$
$\langle \textit{proof}\rangle$

lemma *polarity-spec'*:
assumes $\langle \textit{no-dup}\ M\rangle$
shows
$\langle \textit{polarity}\ M\ L = \textit{None} \longleftrightarrow \textit{undefined-lit}\ M\ L\rangle$ **and**
$\langle \textit{polarity}\ M\ L = \textit{Some True} \longleftrightarrow L \in \textit{lits-of-l}\ M\rangle$ **and**
$\langle \textit{polarity}\ M\ L = \textit{Some False} \longleftrightarrow -L \in \textit{lits-of-l}\ M\rangle$
$\langle \textit{proof}\rangle$

definition *find-unwatched-l* **where**
$\langle \textit{find-unwatched-l}\ M\ C = \textit{SPEC}\ (\lambda(\textit{found}).$
$(\textit{found} = \textit{None} \longleftrightarrow (\forall L{\in}\textit{set}\ (\textit{unwatched-l}\ C).\ -L \in \textit{lits-of-l}\ M)) \wedge$
$(\forall j.\ \textit{found} = \textit{Some}\ j \longrightarrow (j < \textit{length}\ C \wedge (\textit{undefined-lit}\ M\ (C!j) \vee C!j \in \textit{lits-of-l}\ M) \wedge j \ge 2)))\rangle$

definition *set-conflict-l* :: $\langle 'v\ \textit{clause-l} \Rightarrow 'v\ \textit{twl-st-l} \Rightarrow 'v\ \textit{twl-st-l}\rangle$ **where**
$\langle \textit{set-conflict-l} = (\lambda C\ (M, N, D, NE, UE, WS, Q).\ (M, N, \textit{Some}\ (\textit{mset}\ C), NE, UE, \{\#\}, \{\#\}))\rangle$

definition *propagate-lit-l* :: $\langle 'v\ \textit{literal} \Rightarrow \textit{nat} \Rightarrow \textit{nat} \Rightarrow 'v\ \textit{twl-st-l} \Rightarrow 'v\ \textit{twl-st-l}\rangle$ **where**
$\langle \textit{propagate-lit-l} = (\lambda L'\ C\ i\ (M, N, D, NE, UE, WS, Q).$
$\textit{let}\ N = N(C \hookrightarrow (\textit{swap}\ (N \propto C)\ 0\ (\textit{Suc}\ 0 - i)))\ \textit{in}$
$(\textit{Propagated}\ L'\ C\ \#\ M, N, D, NE, UE, WS, \textit{add-mset}\ (-L')\ Q))\rangle$

definition *update-clause-l* :: $\langle \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat} \Rightarrow 'v\ \textit{twl-st-l} \Rightarrow 'v\ \textit{twl-st-l nres}\rangle$ **where**
$\langle \textit{update-clause-l} = (\lambda C\ i\ f\ (M, N, D, NE, UE, WS, Q).\ \textit{do}\ \{$
$\textit{let}\ N' = N\ (C \hookrightarrow (\textit{swap}\ (N{\propto}C)\ i\ f));$
$\textit{RETURN}\ (M, N', D, NE, UE, WS, Q)$
$\})\rangle$

definition *unit-propagation-inner-loop-body-l-inv*
:: $\langle 'v\ \textit{literal} \Rightarrow \textit{nat} \Rightarrow 'v\ \textit{twl-st-l} \Rightarrow \textit{bool}\rangle$
where
$\langle \textit{unit-propagation-inner-loop-body-l-inv}\ L\ C\ S \longleftrightarrow$
$(\exists S'.\ (\textit{set-clauses-to-update-l}\ (\textit{clauses-to-update-l}\ S + \{\#C\#\})\ S, S') \in \textit{twl-st-l}\ (\textit{Some}\ L) \wedge$
$\textit{twl-struct-invs}\ S' \wedge$
$\textit{twl-stgy-invs}\ S' \wedge$
$C \in\# \ \textit{dom-m}\ (\textit{get-clauses-l}\ S) \wedge$

$C > 0 \wedge$
$0 < \textit{length}\ (\textit{get-clauses-l}\ S \propto C) \wedge$
$\textit{no-dup}\ (\textit{get-trail-l}\ S) \wedge$
$(\textit{if}\ (\textit{get-clauses-l}\ S \propto C)\ !\ 0 = L\ \textit{then}\ 0\ \textit{else}\ 1) < \textit{length}\ (\textit{get-clauses-l}\ S \propto C) \wedge$
$1 - (\textit{if}\ (\textit{get-clauses-l}\ S \propto C)\ !\ 0 = L\ \textit{then}\ 0\ \textit{else}\ 1) < \textit{length}\ (\textit{get-clauses-l}\ S \propto C) \wedge$
$L \in \textit{set}\ (\textit{watched-l}\ (\textit{get-clauses-l}\ S \propto C)) \wedge$
$\textit{get-conflict-l}\ S = \textit{None}$
)
›

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*);
  $K \leftarrow \textit{SPEC}(\lambda K.\ K \in \textit{set}\ (\textit{get-clauses-l}\ S \propto C))$;
  let val-K = *polarity* (*get-trail-l S*) *K*;
  if val-K = *Some True then RETURN S*
  else do {
    let i = (*if* ($\textit{get-clauses-l}\ S \propto C$) ! *0* = *L then 0 else 1*);
    let L' = ($\textit{get-clauses-l}\ S \propto C$) ! (*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*) ($\textit{get-clauses-l}\ S \propto C$);
      case f of
        None ⇒
          if val-L' = *Some False*
          then RETURN (*set-conflict-l* ($\textit{get-clauses-l}\ S \propto C$) *S*)
          else RETURN (*propagate-lit-l* L' *C i S*)
      | *Some f* ⇒ *do* {
          ASSERT(*f* < *length* ($\textit{get-clauses-l}\ S \propto C$));
          let K = ($\textit{get-clauses-l}\ S \propto C$)!*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
‹$(f\ S) \leq \textit{SPEC}(\lambda S'.\ Q\ S')$› **and**
‹$y \leq \Downarrow \{(S, S').\ P\ S\ S'\}\ (f\ S)$›
shows ‹$y \leq \Downarrow \{(S, S').\ P\ S\ S' \wedge Q\ S'\}\ (f\ S)$›
⟨*proof*⟩

lemma *clauses-tuple*[*simp*]:
‹$\textit{cdcl}_W$*-restart-mset.clauses* (*M*, {#*f x* . *x* ∈# *init-clss-l N*#} + *NE*,
{#*f x* . *x* ∈# *learned-clss-l N*#} + *UE*, *D*) = {#*f x*. *x* ∈# *all-clss-l N*#} + *NE* + *UE*›
⟨*proof*⟩

lemma *valid-enqueued-alt-simps*[*simp*]:

‹*valid-enqueued* $S \longleftrightarrow$
$(\forall (L, C) \in\# \textit{clauses-to-update } S.\ L \in\# \textit{watched } C \wedge C \in\# \textit{get-clauses } S \wedge$
$-L \in \textit{lits-of-l } (\textit{get-trail } S) \wedge \textit{get-level } (\textit{get-trail } S)\ L = \textit{count-decided } (\textit{get-trail } S)) \wedge$
$(\forall L \in\# \textit{literals-to-update } S.$
$-L \in \textit{lits-of-l } (\textit{get-trail } S) \wedge \textit{get-level } (\textit{get-trail } S)\ L = \textit{count-decided } (\textit{get-trail } S))$›
⟨*proof*⟩

declare *valid-enqueued.simps*[*simp del*]

lemma *set-clauses-simp*[*simp*]:
‹$f\ `\ \{a.\ a \in\# \textit{ran-m } N \wedge \neg \textit{ snd } a\} \cup f\ `\ \{a.\ a \in\# \textit{ran-m } N \wedge \textit{snd } a\} \cup A =$
$f\ `\ \{a.\ a \in\# \textit{ran-m } N\} \cup A$›
⟨*proof*⟩

lemma *init-clss-l-clause-upd*:
‹$C \in\# \textit{dom-m } N \Longrightarrow \textit{irred } N\ C \Longrightarrow$
$\textit{init-clss-l } (N(C \hookrightarrow C')) =$
$\textit{add-mset } (C', \textit{irred } N\ C)\ (\textit{remove1-mset } (N \propto C, \textit{irred } N\ C)\ (\textit{init-clss-l } N))$›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd*:
‹$C \in\# \textit{dom-m } N \Longrightarrow \textit{irred } N\ C \Longrightarrow$
$\textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{True})\ N) =$
$\textit{add-mset } (C', \textit{irred } N\ C)\ (\textit{remove1-mset } (N \propto C, \textit{irred } N\ C)\ (\textit{init-clss-l } N))$›
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd*:
‹$C \in\# \textit{dom-m } N \Longrightarrow \neg \textit{irred } N\ C \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) =$
$\textit{add-mset } (C', \textit{irred } N\ C)\ (\textit{remove1-mset } (N \propto C, \textit{irred } N\ C)\ (\textit{learned-clss-l } N))$›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-irrel*: ‹$C \in\# \textit{dom-m } N \Longrightarrow \neg \textit{irred } N\ C \Longrightarrow$
$\textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) = \textit{init-clss-l } N$›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-irrel-notin*: ‹$C \notin\# \textit{dom-m } N \Longrightarrow$
$\textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) = \textit{init-clss-l } N$›
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd-irrel*: ‹$C \in\# \textit{dom-m } N \Longrightarrow \textit{irred } N\ C \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmupd } C\ (C', \textit{True})\ N) = \textit{learned-clss-l } N$›
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd-notin*: ‹$C \notin\# \textit{dom-m } N \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) = \textit{add-mset } (C', \textit{False})\ (\textit{learned-clss-l } N)$›
⟨*proof*⟩

lemma *in-ran-mf-clause-inI*[*intro*]:
‹$C \in\# \textit{dom-m } N \Longrightarrow i = \textit{irred } N\ C \Longrightarrow (N \propto C, i) \in\# \textit{ran-m } N$›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-notin*:
‹$C \notin\# \textit{dom-m } N \Longrightarrow \textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{True})\ N) =$
$\textit{add-mset } (C', \textit{True})\ (\textit{init-clss-l } N)$›
⟨*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 *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-clauses-to-update-l* (*clauses-to-update-l* S − {#C#}) S) ≤
⇓ {(S, S''). (S, S'') ∈ *twl-st-l* (*Some* L) ∧ *twl-list-invs* S ∧ *twl-stgy-invs* S'' ∧
twl-struct-invs S''}
(*unit-propagation-inner-loop-body* L (*twl-clause-of* C')
(*set-clauses-to-update* (*clauses-to-update* (S') − {#(L, *twl-clause-of* C')#}) S'))›
(**is** ‹*?A* ≤ ⇓ - *?B*›)
⟨*proof*⟩

lemma *unit-propagation-inner-loop-body-l2*:
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-clauses-to-update-l* (*clauses-to-update-l* S − {#C#}) S),
unit-propagation-inner-loop-body L (*twl-clause-of* (*get-clauses-l* $S \propto C$))
(*set-clauses-to-update*
(*remove1-mset* (L, *twl-clause-of* (*get-clauses-l* $S \propto C$))
(*clauses-to-update* S')) S'))
∈ ⟨{(S, S'). (S, S') ∈ *twl-st-l* (*Some* L) ∧ *twl-list-invs* S ∧ *twl-stgy-invs* S' ∧
twl-struct-invs S'}⟩*nres-rel*›
⟨*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**
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L = (\lambda(S,\ n).$
$(\exists S'.\ (S,\ S') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}struct\text{-}invs\ S' \wedge twl\text{-}stgy\text{-}invs\ S' \wedge$
$twl\text{-}list\text{-}invs\ S \wedge (clauses\text{-}to\text{-}update\ S' \neq \{\#\} \vee n > 0 \longrightarrow get\text{-}conflict\ S' = None) \wedge$
$-L \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ S)))$›

definition *unit-propagation-inner-loop-body-l-with-skip* **where**
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L = (\lambda(S,\ n).\ do\ \{$
$ASSERT\ (clauses\text{-}to\text{-}update\text{-}l\ S \neq \{\#\} \vee n > 0);$
$ASSERT(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L\ (S,\ n));$
$b \leftarrow SPEC(\lambda b.\ (b \longrightarrow n > 0) \wedge (\neg b \longrightarrow clauses\text{-}to\text{-}update\text{-}l\ S \neq \{\#\}));$
$if\ \neg b\ then\ do\ \{$
$ASSERT\ (clauses\text{-}to\text{-}update\text{-}l\ S \neq \{\#\});$
$(S',\ C) \leftarrow select\text{-}from\text{-}clauses\text{-}to\text{-}update\ S;$
$T \leftarrow unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\ L\ C\ S';$
$RETURN\ (T,\ if\ get\text{-}conflict\text{-}l\ T = None\ then\ n\ else\ 0)$
$\}\ else\ RETURN\ (S,\ n{-}1)$
$\})$›

definition *unit-propagation-inner-loop-l* :: ‹$'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}l \Rightarrow 'v\ twl\text{-}st\text{-}l\ nres$› **where**
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\ L\ S_0 = do\ \{$
$n \leftarrow SPEC(\lambda\text{-}::nat.\ True);$
$(S,\ n) \leftarrow WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L}$
$(\lambda(S,\ n).\ clauses\text{-}to\text{-}update\text{-}l\ S \neq \{\#\} \vee n > 0)$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L)$
$(S_0,\ n);$
$RETURN\ S$
$\}$›

lemma *set-mset-clauses-to-update-l-set-mset-clauses-to-update-spec*:
assumes ‹$(S,\ S') \in twl\text{-}st\text{-}l\ (Some\ L)$›
shows
‹$RES\ (set\text{-}mset\ (clauses\text{-}to\text{-}update\text{-}l\ S)) \le \Downarrow \{(C,\ (L',\ C')).\ L' = L \wedge$
$C' = twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C)\}$
$(RES\ (set\text{-}mset\ (clauses\text{-}to\text{-}update\ S')))$›
⟨*proof*⟩

lemma *refine-add-inv*:
fixes f :: ‹$'a \Rightarrow 'a\ nres$› **and** f' :: ‹$'b \Rightarrow 'b\ nres$› **and** h :: ‹$'b \Rightarrow 'a$›
assumes

‹$(f', f) \in \{(S, S').\ S' = h\ S \wedge R\ S\} \to \langle\{(T, T').\ T' = h\ T \wedge P'\ T\}\rangle$ *nres-rel*›
(**is** ‹- $\in$ *?R* $\to \langle\{(T, T').$ *?H* $T\ T' \wedge P'\ T\}\rangle$ *nres-rel*›)
assumes
‹$\bigwedge S.\ R\ S \Longrightarrow f\ (h\ S) \le SPEC\ (\lambda T.\ Q\ T)$›
shows
‹$(f', f) \in$ *?R* $\to \langle\{(T, T').$ *?H* $T\ T' \wedge P'\ T \wedge Q\ (h\ T)\}\rangle$ *nres-rel*›
⟨*proof*⟩

lemma *refine-add-inv-generalised*:
fixes f :: ‹$'a \Rightarrow 'b$ *nres*› **and** f' :: ‹$'c \Rightarrow 'd$ *nres*›
assumes
‹$(f', f) \in A \to_f \langle B\rangle$ *nres-rel*›
assumes
‹$\bigwedge S\ S'.\ (S, S') \in A \Longrightarrow f\ S' \le RES\ C$›
shows
‹$(f', f) \in A \to_f \langle\{(T, T').\ (T, T') \in B \wedge T' \in C\}\rangle$ *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-rel*› (**is** ‹- $\in$ *?R* $\to \langle\{(S, S').$ *?H* $S\ S' \wedge P'\ S\}\rangle$ *nres-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-rel*›
⟨*proof*⟩

lemma *clauses-to-update-l-empty-tw-st-of-Some-None*[*simp*]:
‹*clauses-to-update-l* $S = \{\#\} \Longrightarrow (S, S') \in$ *twl-st-l* (*Some* L) $\longleftrightarrow (S, S') \in$ *twl-st-l None*›
⟨*proof*⟩

lemma *cdcl-twl-cp-in-trail-stays-in*:
‹*cdcl-twl-cp*** S' *aa* $\Longrightarrow -$ *x1* $\in$ *lits-of-l* (*get-trail* S') $\Longrightarrow -$ *x1* $\in$ *lits-of-l* (*get-trail aa*)›
⟨*proof*⟩

lemma *cdcl-twl-cp-in-trail-stays-in-l*:
‹(*x2*, S') $\in$ *twl-st-l* (*Some x1*) $\Longrightarrow$ *cdcl-twl-cp*** S' *aa* $\Longrightarrow -$ *x1* $\in$ *lits-of-l* (*get-trail-l x2*) $\Longrightarrow$
(*a*, *aa*) $\in$ *twl-st-l* (*Some x1*) $\Longrightarrow -$ *x1* $\in$ *lits-of-l* (*get-trail-l a*)›
⟨*proof*⟩

lemma *unit-propagation-inner-loop-l*:
‹(*uncurry unit-propagation-inner-loop-l*, *unit-propagation-inner-loop*) $\in$
$\{((L, S), S').\ (S, S') \in$ *twl-st-l* (*Some* L) $\wedge$ *twl-struct-invs* $S' \wedge$
twl-stgy-invs $S' \wedge$ *twl-list-invs* $S \wedge -L \in$ *lits-of-l* (*get-trail-l* S)$\} \to_f$
$\langle\{(T, T').\ (T, T') \in$ *twl-st-l None* $\wedge$ *clauses-to-update-l* $T = \{\#\} \wedge$
twl-list-invs $T \wedge$ *twl-struct-invs* $T' \wedge$ *twl-stgy-invs* $T'\}\rangle$ *nres-rel*›
(**is** ‹*?unit-prop-inner* $\in$ *?A* $\to_f \langle$*?B*$\rangle$*nres-rel*›)
⟨*proof*⟩

definition *clause-to-update* :: ‹$'v$ *literal* $\Rightarrow 'v$ *twl-st-l* $\Rightarrow 'v$ *clauses-to-update-l*›**where**
‹*clause-to-update* L S =
filter-mset
(λC::*nat*. $L \in$ *set* (*watched-l* (*get-clauses-l* $S \propto C$)))
(*dom-m* (*get-clauses-l* S))›

lemma *distinct-mset-clause-to-update*: ‹*distinct-mset* (*clause-to-update* L C)›
⟨*proof*⟩

lemma *in-clause-to-updateD*: ‹$b \in\#$ *clause-to-update* L' $T \Longrightarrow b \in\#$ *dom-m* (*get-clauses-l* T)›
⟨*proof*⟩

lemma *in-clause-to-update-iff*:
‹$C \in\#$ *clause-to-update* L $S \longleftrightarrow$
$C \in\#$ *dom-m* (*get-clauses-l* S) $\wedge$ $L \in$ *set* (*watched-l* (*get-clauses-l* $S \propto C$))›
⟨*proof*⟩

definition *select-and-remove-from-literals-to-update* :: ‹$'v$ *twl-st-l* $\Rightarrow$
($'v$ *twl-st-l* $\times$ $'v$ *literal*) *nres*› **where**
‹*select-and-remove-from-literals-to-update* S = *SPEC*(λ(S', L). $L \in\#$ *literals-to-update-l* S $\wedge$
S' = *set-clauses-to-update-l* (*clause-to-update* L S)
(*set-literals-to-update-l* (*literals-to-update-l* S − {#L#}) S))›

definition *unit-propagation-outer-loop-l-inv* **where**
‹*unit-propagation-outer-loop-l-inv* $S \longleftrightarrow$
($\exists$ 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* $\Rightarrow$ $'v$ *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) ← *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)) $\longleftrightarrow$

$((\forall x{\in}\#WS.\ case\ x\ of\ C \Rightarrow C \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}l\ T)) \land$
$distinct\text{-}mset\ WS)$›
⟨*proof*⟩

lemma *unit-propagation-outer-loop-l-spec*:
‹$(unit\text{-}propagation\text{-}outer\text{-}loop\text{-}l,\ unit\text{-}propagation\text{-}outer\text{-}loop) \in$
$\{(S,\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ None \land twl\text{-}struct\text{-}invs\ S' \land$
$twl\text{-}stgy\text{-}invs\ S' \land twl\text{-}list\text{-}invs\ S \land clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\} \land$
$get\text{-}conflict\text{-}l\ S = None\} \rightarrow_f$
$\langle\{(T,\ T').\ (T,\ T') \in twl\text{-}st\text{-}l\ None \land$
$(twl\text{-}list\text{-}invs\ T \land twl\text{-}struct\text{-}invs\ T' \land twl\text{-}stgy\text{-}invs\ T' \land$
$clauses\text{-}to\text{-}update\text{-}l\ T = \{\#\}) \land$
$literals\text{-}to\text{-}update\ T' = \{\#\} \land clauses\text{-}to\text{-}update\ T' = \{\#\} \land$
$no\text{-}step\ cdcl\text{-}twl\text{-}cp\ T'\}\rangle\ nres\text{-}rel$›
(**is** ‹$\text{-} \in\ ?R \rightarrow_f\ ?I$› **is** ‹$\text{-} \in \text{-} \rightarrow_f \langle ?B\rangle\ nres\text{-}rel$›)
⟨*proof*⟩

lemma *get-conflict-l-get-conflict-state-spec*:
assumes ‹$(S,\ S') \in twl\text{-}st\text{-}l\ None$› **and** ‹$twl\text{-}list\text{-}invs\ S$› **and** ‹$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}$›
shows ‹$((False,\ S),\ (False,\ S'))$
$\in \{((brk,\ S),\ (brk',\ S')).\ brk = brk' \land (S,\ S') \in twl\text{-}st\text{-}l\ None \land twl\text{-}list\text{-}invs\ S \land$
$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}\}$›
⟨*proof*⟩

fun *lit-and-ann-of-propagated* **where**
‹$lit\text{-}and\text{-}ann\text{-}of\text{-}propagated\ (Propagated\ L\ C) = (L,\ C)$› |
‹$lit\text{-}and\text{-}ann\text{-}of\text{-}propagated\ (Decided\ \text{-}) = undefined$›
— we should never call the function in that context

definition *tl-state-l* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow 'v\ twl\text{-}st\text{-}l$› **where**
‹$tl\text{-}state\text{-}l = (\lambda(M,\ N,\ D,\ NE,\ UE,\ WS,\ Q).\ (tl\ M,\ N,\ D,\ NE,\ UE,\ WS,\ Q))$›

definition *resolve-cls-l'* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow nat \Rightarrow 'v\ literal \Rightarrow 'v\ clause$› **where**
‹$resolve\text{-}cls\text{-}l'\ S\ C\ L =$
$remove1\text{-}mset\ (-L)\ (the\ (get\text{-}conflict\text{-}l\ S) \cup\#\ mset\ (tl\ (get\text{-}clauses\text{-}l\ S \propto C)))$›

definition *update-confl-tl-l* :: ‹$nat \Rightarrow 'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}l \Rightarrow bool \times 'v\ twl\text{-}st\text{-}l$› **where**
‹$update\text{-}confl\text{-}tl\text{-}l = (\lambda C\ L\ (M,\ N,\ D,\ NE,\ UE,\ WS,\ Q).$
$let\ D = resolve\text{-}cls\text{-}l'\ (M,\ N,\ D,\ NE,\ UE,\ WS,\ Q)\ C\ L\ in$
$(False,\ (tl\ M,\ N,\ Some\ D,\ NE,\ UE,\ WS,\ Q)))$›

definition *skip-and-resolve-loop-inv-l* **where**
‹$skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\text{-}l\ S_0\ brk\ S \longleftrightarrow$
$(\exists S'\ S_0'.\ (S,\ S') \in twl\text{-}st\text{-}l\ None \land (S_0,\ S_0') \in twl\text{-}st\text{-}l\ None \land$
$skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\ S_0'\ (brk,\ S') \land$
$twl\text{-}list\text{-}invs\ S \land clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\} \land$
$(\neg is\text{-}decided\ (hd\ (get\text{-}trail\text{-}l\ S)) \longrightarrow mark\text{-}of\ (hd(get\text{-}trail\text{-}l\ S)) > 0))$›

definition *skip-and-resolve-loop-l* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow 'v\ twl\text{-}st\text{-}l\ nres$› **where**
‹$skip\text{-}and\text{-}resolve\text{-}loop\text{-}l\ S_0 =$
$do\ \{$
$ASSERT(get\text{-}conflict\text{-}l\ S_0 \neq None);$
$(\text{-},\ S) \leftarrow$
$WHILE_T{}^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\text{-}l\ S_0\ brk\ S}$
$(\lambda(brk,\ S).\ \neg brk \land \neg is\text{-}decided\ (hd\ (get\text{-}trail\text{-}l\ S)))$

$(\lambda(\text{-}, S).$
do {
let $D' = $ *the* (*get-conflict-l* S);
let $(L, C) = $ *lit-and-ann-of-propagated* (*hd* (*get-trail-l* S));
if $-L \notin\# D'$ *then*
do {*RETURN* (*False*, *tl-state-l* S)}
else
if *get-maximum-level* (*get-trail-l* S) (*remove1-mset* $(-L)$ D') = *count-decided* (*get-trail-l* S)
then
do {*RETURN* (*update-confl-tl-l* C L S)}
else
do {*RETURN* (*True*, S)}
}
)
(*False*, S_0);
RETURN S
}
›

context
begin

private lemma *skip-and-resolve-l-refines*:
‹((*brkS*), *brk'S'*) ∈ {((*brk*, S), *brk'*, S'). *brk* = *brk'* ∧ (S, S') ∈ *twl-st-l None* ∧
twl-list-invs S ∧ *clauses-to-update-l* S = {#}} ⟹
brkS = (*brk*, S) ⟹ *brk'S'* = (*brk'*, S') ⟹
((*False*, *tl-state-l* S), *False*, *tl-state* S') ∈ {((*brk*, S), *brk'*, S'). *brk* = *brk'* ∧
(S, S') ∈ *twl-st-l None* ∧ *twl-list-invs* S ∧ *clauses-to-update-l* S = {#}}›
⟨*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'). $L = L' \wedge C > 0 \wedge$
C' = *mset* (*get-clauses-l* $S \propto C$)}› **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*›
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* ∉# *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* = $(\lambda(M, N, D, NE, UE, WS, Q)\ L.$
$SPEC(\lambda L'.\ L' \in\# \mathit{the}\ D - \{\#-L\#\} \wedge \mathit{get\text{-}level}\ M\ L' = \mathit{get\text{-}maximum\text{-}level}\ M\ (\mathit{the}\ D - \{\#-L\#\})))$›

definition *ex-decomp-of-max-lvl* :: ‹$('v, nat)$ *ann-lits* ⇒ $'v$ *cconflict* ⇒ $'v$ *literal* ⇒ *bool*› **where**
‹*ex-decomp-of-max-lvl* $M\ D\ L \longleftrightarrow$
$(\exists K\ M1\ M2.\ (\mathit{Decided}\ K\ \#\ M1,\ M2) \in \mathit{set}\ (\mathit{get\text{-}all\text{-}ann\text{-}decomposition}\ M) \wedge$
$\mathit{get\text{-}level}\ M\ K = \mathit{get\text{-}maximum\text{-}level}\ M\ (\mathit{remove1\text{-}mset}\ (-L)\ (\mathit{the}\ D)) + 1)$›

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* $D = SPEC(\lambda D'.\ D = \mathit{mset}\ D')$›

fun *extract-shorter-conflict-l* :: ‹$'v$ *twl-st-l* ⇒ $'v$ *twl-st-l nres*›
where
‹*extract-shorter-conflict-l* $(M, N, D, NE, UE, WS, Q) = SPEC(\lambda S.$
$\exists D'.\ D' \subseteq\# \mathit{the}\ D \wedge S = (M, N, \mathit{Some}\ D', NE, UE, WS, Q) \wedge$
clause ‘# *twl-clause-of* ‘# *ran-mf* $N + NE + UE \models pm\ D' \wedge -(\mathit{lit\text{-}of}\ (\mathit{hd}\ M)) \in\# D')$›

declare *extract-shorter-conflict-l.simps*[*simp del*]
lemmas *extract-shorter-conflict-l-def* = *extract-shorter-conflict-l.simps*

lemma *extract-shorter-conflict-l-alt-def*:
‹*extract-shorter-conflict-l* $S = SPEC(\lambda T.$
$\exists D'.\ D' \subseteq\# \mathit{the}\ (\mathit{get\text{-}conflict\text{-}l}\ S) \wedge \mathit{equality\text{-}except\text{-}conflict\text{-}l}\ S\ T \wedge$
$\mathit{get\text{-}conflict\text{-}l}\ T = \mathit{Some}\ D' \wedge$
clause ‘# *twl-clause-of* ‘# *ran-mf* (*get-clauses-l* S) + *get-unit-clauses-l* $S \models pm\ D' \wedge$
$-\mathit{lit\text{-}of}\ (\mathit{hd}\ (\mathit{get\text{-}trail\text{-}l}\ S)) \in\# D')$›
⟨*proof*⟩

definition *backtrack-l-inv* **where**
‹*backtrack-l-inv* $S \longleftrightarrow$
$(\exists S'.\ (S, S') \in \mathit{twl\text{-}st\text{-}l}\ \mathit{None} \wedge$
$\mathit{get\text{-}trail\text{-}l}\ S \neq [] \wedge$
no-step cdcl_W*-restart-mset.skip* (state_W*-of* S')∧
no-step cdcl_W*-restart-mset.resolve* (state_W*-of* S') ∧
$\mathit{get\text{-}conflict\text{-}l}\ S \neq \mathit{None} \wedge$
twl-struct-invs S' ∧
twl-stgy-invs S' ∧
twl-list-invs S ∧
$\mathit{get\text{-}conflict\text{-}l}\ S \neq \mathit{Some}\ \{\#\})$
›

definition *get-fresh-index* :: ‹$'v$ *clauses-l* ⇒ *nat nres*› **where**
‹*get-fresh-index* $N = SPEC(\lambda i.\ i > 0 \wedge i \notin\# \mathit{dom\text{-}m}\ N)$›

definition *propagate-bt-l* :: ‹$'v$ *literal* ⇒ $'v$ *literal* ⇒ $'v$ *twl-st-l* ⇒ $'v$ *twl-st-l nres*› **where**
‹*propagate-bt-l* = $(\lambda L\ L'\ (M, N, D, NE, UE, WS, Q).$ *do* {
$D'' \leftarrow$ *list-of-mset* (*the* D);
$i \leftarrow$ *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 (λL.
(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* = ($\lambda L'$ (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** ‹$f \le \Downarrow R\ g$› **for** f :: ‹'*a nres*› **and** R :: ‹('*a* × '*b*) *set*› **and**
g :: ‹'*b nres*› ⇒
‹*match premises in*
I[*thin*,*uncurry*]: ‹$f \le \Downarrow R'\ g$› *for* R' :: ‹('*a* × '*b*) *set*›
⇒ ‹*rule refinement-trans-long*[*of f f g g R' R*, *OF refl refl - I*]›
| I[*thin*,*uncurry*]: ‹- ⟹ $f \le \Downarrow R'\ g$› *for* R' :: ‹('*a* × '*b*) *set*›
⇒ ‹*rule refinement-trans-long*[*of f f g g R' R*, *OF refl refl - I*]›
›)

lemma *decide-l-or-skip-spec*:
‹(*decide-l-or-skip*, *decide-or-skip*) ∈
{(S, S'). (S, S') ∈ *twl-st-l None* ∧ *get-conflict-l S* = *None* ∧
clauses-to-update-l S = {#} ∧ *literals-to-update-l S* = {#} ∧ *no-step cdcl-twl-cp S'* ∧
twl-struct-invs S' ∧ *twl-stgy-invs S'* ∧ *twl-list-invs S*} $\to_f$
⟨{((*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* $\to_f$ ⟨*?S*⟩*nres-rel*›)
⟨*proof*⟩

lemma *refinement-trans-eq*:
‹$A = A' \Longrightarrow B = B' \Longrightarrow R' = R \Longrightarrow A \le \Downarrow R\ B \Longrightarrow A' \le \Downarrow R'\ B'$›
⟨*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*:
‹($\bigwedge$*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* ≤ ⇓ *R' g* ⟹ *R'* ⊆ *R* ⟹ *f* ≤ ⇓ *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*} $\to_f$
⟨{((*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* $\to_f$ *?I*› **is** ‹ - ∈ *?R* $\to_f$ ⟨*?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* S_0 =
do {
do {
(*brk*, *T*) ← $WHILE_T{}^{cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}inv\ S_0}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*).
do {
T ← *unit-propagation-outer-loop-l S*;

$cdcl\text{-}twl\text{-}o\text{-}prog\text{-}l\ T$
})
($False$, S_0);
$RETURN\ T$
}
}
›

lemma *cdcl-twl-stgy-prog-l-spec*:
‹($cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l$, $cdcl\text{-}twl\text{-}stgy\text{-}prog$) $\in$
$\{(S, S'). (S, S') \in twl\text{-}st\text{-}l\ None\ \wedge twl\text{-}list\text{-}invs\ S\ \wedge$
$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\} \wedge$
$twl\text{-}struct\text{-}invs\ S' \wedge twl\text{-}stgy\text{-}invs\ S'\} \rightarrow_f$
$\langle\{(T, T'). (T, T') \in \{(T, T'). (T, T') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ T\ \wedge$
$twl\text{-}struct\text{-}invs\ T' \wedge twl\text{-}stgy\text{-}invs\ T'\} \wedge True\}\rangle\ nres\text{-}rel$›
(**is** ‹ - $\in$ $?R \rightarrow_f\ ?I$› **is** ‹ - $\in$ $?R \rightarrow_f \langle ?J\rangle nres\text{-}rel$›)
⟨*proof*⟩

lemma *refine-pair-to-SPEC*:
fixes f :: ‹$'s \Rightarrow\ 's\ nres$› **and** g :: ‹$'b \Rightarrow\ 'b\ nres$›
assumes ‹$(f, g) \in \{(S, S'). (S, S') \in H \wedge R\ S\ S'\} \rightarrow_f \langle\{(S, S'). (S, S') \in H' \wedge P'\ S\}\rangle nres\text{-}rel$›
(**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**
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}pre\ S\ S' \longleftrightarrow$
$((S, S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}struct\text{-}invs\ S' \wedge twl\text{-}stgy\text{-}invs\ S'\ \wedge$
$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\} \wedge get\text{-}conflict\text{-}l\ S = None \wedge twl\text{-}list\text{-}invs\ S)$›

lemma *cdcl-twl-stgy-prog-l-spec-final*:
assumes
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}pre\ S\ S'$›
shows
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\ S \leq \Downarrow (twl\text{-}st\text{-}l\ None)\ (conclusive\text{-}TWL\text{-}run\ S')$›
⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-l-spec-final'*:
assumes
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}pre\ S\ S'$›
shows
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\ S \leq \Downarrow \{(S, T). (S, T) \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S\ \wedge$
$twl\text{-}struct\text{-}invs\ S' \wedge twl\text{-}stgy\text{-}invs\ S'\}\ (conclusive\text{-}TWL\text{-}run\ S')$›
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-break-l* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow\ 'v\ twl\text{-}st\text{-}l\ nres$› **where**
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}break\text{-}l\ S_0 =$
do {
$b \leftarrow SPEC(\lambda\text{-}.\ True)$;
$(b, brk, T) \leftarrow WHILE_T{}^{\lambda(b,\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}inv\ S_0\ S}$
$(\lambda(b, brk, \text{-}).\ b \wedge \neg brk)$
$(\lambda(\text{-}, brk, S).$ *do* {
$T \leftarrow unit\text{-}propagation\text{-}outer\text{-}loop\text{-}l\ S$;
$T \leftarrow cdcl\text{-}twl\text{-}o\text{-}prog\text{-}l\ T$;
$b \leftarrow SPEC(\lambda\text{-}.\ True)$;

RETURN (*b*, *T*)
})
(*b*, *False*, S_0);
if brk then RETURN T
else cdcl-twl-stgy-prog-l T
}›

lemma *cdcl-twl-stgy-prog-break-l-spec*:
‹(*cdcl-twl-stgy-prog-break-l*, *cdcl-twl-stgy-prog-break*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#} ∧
twl-struct-invs S' ∧ *twl-stgy-invs S'*} $\rightarrow_f$
⟨{(*T*, *T'*). (*T*, *T'*) ∈ {(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs T* ∧
twl-struct-invs T' ∧ *twl-stgy-invs T'*} ∧ *True*}⟩ *nres-rel*›
(**is** ‹ - ∈ *?R* $\rightarrow_f$ *?I*› **is** ‹ - ∈ *?R* $\rightarrow_f$ ⟨*?J*⟩*nres-rel*›)
⟨*proof*⟩

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-Watch-List*
imports *Watched-Literals-List Array-UInt*
begin

Remove notation that coonflicts with *list-update*:

no-notation *Ref.update* (- := - *62*)

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*]

fun *all-blits-are-in-problem* **where**
‹*all-blits-are-in-problem* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*)). (∀ (*i*, *K*)∈#*mset* (*W L*). *K* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))))›

declare *all-blits-are-in-problem.simps*[*simp del*]

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*) ⟷

($\forall L \in\# \; all\text{-}lits\text{-}of\text{-}mm \; (mset \; `\# \; ran\text{-}mf \; N + (NE + UE))$.
($L = K \longrightarrow$
$((\forall (i, K, b)\in\# mset \; (take \; i \; (W \; L) \; @ \; drop \; j \; (W \; L)).\; i \in\# \; dom\text{-}m \; N \longrightarrow K \in set \; (N \propto i) \wedge$
$K \neq L \wedge correctly\text{-}marked\text{-}as\text{-}binary \; N \; (i, K, b)) \wedge$
$(\forall (i, K, b)\in\# mset \; (take \; i \; (W \; L) \; @ \; drop \; j \; (W \; L)).\; b \longrightarrow i \in\# \; dom\text{-}m \; N) \wedge$
$filter\text{-}mset \; (\lambda i.\; i \in\# \; dom\text{-}m \; N) \; (fst \; `\# \; mset \; (take \; i \; (W \; L) \; @ \; drop \; j \; (W \; L))) = clause\text{-}to\text{-}update$
$L \; (M, N, D, NE, UE, \{\#\}, \{\#\}))) \wedge$
$(L \neq K \longrightarrow$
$((\forall (i, K, b)\in\# mset \; (W \; L).\; i \in\# \; dom\text{-}m \; N \longrightarrow K \in set \; (N \propto i) \wedge K \neq L \wedge correctly\text{-}marked\text{-}as\text{-}binary$
$N \; (i, K, b)) \wedge$
$(\forall (i, K, b)\in\# mset \; (W \; L).\; b \longrightarrow i \in\# \; dom\text{-}m \; N) \wedge$
$filter\text{-}mset \; (\lambda i.\; i \in\# \; dom\text{-}m \; N) \; (fst \; `\# \; mset \; (W \; L)) = clause\text{-}to\text{-}update \; L \; (M, N, D, NE, UE,$
$\{\#\}, \{\#\})))))$›

fun *correct-watching* :: ‹$'v \; twl\text{-}st\text{-}wl \Rightarrow bool$› **where**
‹$correct\text{-}watching \; (M, N, D, NE, UE, Q, W) \longleftrightarrow$
$(\forall L \in\# \; all\text{-}lits\text{-}of\text{-}mm \; (mset \; `\# \; ran\text{-}mf \; N + (NE + UE))$.
$(\forall (i, K, b)\in\# mset \; (W \; L).\; i \in\# \; dom\text{-}m \; N \longrightarrow K \in set \; (N \propto i) \wedge K \neq L \wedge correctly\text{-}marked\text{-}as\text{-}binary$
$N \; (i, K, b)) \wedge$
$(\forall (i, K, b)\in\# mset \; (W \; L).\; b \longrightarrow i \in\# \; dom\text{-}m \; N) \wedge$
$filter\text{-}mset \; (\lambda i.\; i \in\# \; dom\text{-}m \; N) \; (fst \; `\# \; mset \; (W \; L)) = clause\text{-}to\text{-}update \; L \; (M, N, D, NE, UE,$
$\{\#\}, \{\#\}))$›

declare *correct-watching.simps*[*simp del*]

lemma *correct-watching-except-correct-watching*:
assumes
j: ‹$j \geq length \; (W \; K)$› **and**
corr: ‹$correct\text{-}watching\text{-}except \; i \; j \; K \; (M, N, D, NE, UE, Q, W)$›
shows ‹$correct\text{-}watching \; (M, N, D, NE, UE, Q, W(K := take \; i \; (W \; K)))$›
⟨*proof*⟩

fun *watched-by* :: ‹$'v \; twl\text{-}st\text{-}wl \Rightarrow 'v \; literal \Rightarrow 'v \; watched$› **where**
‹$watched\text{-}by \; (M, N, D, NE, UE, Q, W) \; L = W \; L$›

fun *update-watched* :: ‹$'v \; literal \Rightarrow 'v \; watched \Rightarrow 'v \; twl\text{-}st\text{-}wl \Rightarrow 'v \; twl\text{-}st\text{-}wl$› **where**
‹$update\text{-}watched \; L \; WL \; (M, N, D, NE, UE, Q, W) = (M, N, D, NE, UE, Q, W(L{:=} \; WL))$›

lemma *bspec'*: ‹$x \in a \Longrightarrow \forall x{\in}a.\; P \; x \Longrightarrow P \; x$›
⟨*proof*⟩

lemma *correct-watching-exceptD*:
assumes
‹$correct\text{-}watching\text{-}except \; i \; j \; L \; S$› **and**
‹$L \in\# \; all\text{-}lits\text{-}of\text{-}mm$
$(mset \; `\# \; ran\text{-}mf \; (get\text{-}clauses\text{-}wl \; S) + get\text{-}unit\text{-}clauses\text{-}wl \; S)$› **and**
w: ‹$w < length \; (watched\text{-}by \; S \; L)$› ‹$w \geq j$› ‹$fst \; (watched\text{-}by \; S \; L \; ! \; w) \in\# \; dom\text{-}m \; (get\text{-}clauses\text{-}wl \; S)$›
shows ‹$fst \; (snd \; (watched\text{-}by \; S \; L \; ! \; w)) \in set \; (get\text{-}clauses\text{-}wl \; S \propto (fst \; (watched\text{-}by \; S \; L \; ! \; w)))$›
⟨*proof*⟩

declare *correct-watching-except.simps*[*simp del*]

lemma *in-all-lits-of-mm-ain-atms-of-iff*:
‹$L \in\# \; all\text{-}lits\text{-}of\text{-}mm \; N \longleftrightarrow atm\text{-}of \; L \in atms\text{-}of\text{-}mm \; N$›
⟨*proof*⟩

lemma *all-lits-of-mm-union*:
‹$\textit{all-lits-of-mm}\ (M + N) = \textit{all-lits-of-mm}\ M + \textit{all-lits-of-mm}\ N$›
⟨*proof*⟩

definition *all-lits-of-m* :: ‹$'a\ \textit{clause} \Rightarrow 'a\ \textit{literal multiset}$› **where**
‹$\textit{all-lits-of-m}\ Ls = \textit{Pos}\ `\#\ (\textit{atm-of}\ `\#\ Ls) + \textit{Neg}\ `\#\ (\textit{atm-of}\ `\#\ Ls)$›

lemma *all-lits-of-m-empty*[*simp*]: ‹$\textit{all-lits-of-m}\ \{\#\} = \{\#\}$›
⟨*proof*⟩

lemma *all-lits-of-m-empty-iff* [*iff*]: ‹$\textit{all-lits-of-m}\ A = \{\#\} \longleftrightarrow A = \{\#\}$›
⟨*proof*⟩

lemma *in-all-lits-of-m-ain-atms-of-iff*: ‹$L \in\#\ \textit{all-lits-of-m}\ N \longleftrightarrow \textit{atm-of}\ L \in \textit{atms-of}\ N$›
⟨*proof*⟩

lemma *in-clause-in-all-lits-of-m*: ‹$x \in\#\ C \Longrightarrow x \in\#\ \textit{all-lits-of-m}\ C$›
⟨*proof*⟩

lemma *all-lits-of-mm-add-mset*:
‹$\textit{all-lits-of-mm}\ (\textit{add-mset}\ C\ N) = (\textit{all-lits-of-m}\ C) + (\textit{all-lits-of-mm}\ N)$›
⟨*proof*⟩

lemma *all-lits-of-m-add-mset*:
‹$\textit{all-lits-of-m}\ (\textit{add-mset}\ L\ C) = \textit{add-mset}\ L\ (\textit{add-mset}\ (-L)\ (\textit{all-lits-of-m}\ C))$›
⟨*proof*⟩

lemma *all-lits-of-m-union*:
‹$\textit{all-lits-of-m}\ (A + B) = \textit{all-lits-of-m}\ A + \textit{all-lits-of-m}\ B$›
⟨*proof*⟩

lemma *all-lits-of-m-mono*:
‹$D \subseteq\#\ D' \Longrightarrow \textit{all-lits-of-m}\ D \subseteq\#\ \textit{all-lits-of-m}\ D'$›
⟨*proof*⟩

lemma *in-all-lits-of-mm-uminusD*: ‹$x2 \in\#\ \textit{all-lits-of-mm}\ N \Longrightarrow -x2 \in\#\ \textit{all-lits-of-mm}\ N$›
⟨*proof*⟩

lemma *in-all-lits-of-mm-uminus-iff*: ‹$-x2 \in\#\ \textit{all-lits-of-mm}\ N \longleftrightarrow x2 \in\#\ \textit{all-lits-of-mm}\ N$›
⟨*proof*⟩

lemma *all-lits-of-mm-diffD*:
‹$L \in\#\ \textit{all-lits-of-mm}\ (A - B) \Longrightarrow L \in\#\ \textit{all-lits-of-mm}\ A$›
⟨*proof*⟩

lemma *all-lits-of-mm-mono*:
‹$\textit{set-mset}\ A \subseteq \textit{set-mset}\ B \Longrightarrow \textit{set-mset}\ (\textit{all-lits-of-mm}\ A) \subseteq \textit{set-mset}\ (\textit{all-lits-of-mm}\ B)$›
⟨*proof*⟩

fun *st-l-of-wl* :: ‹$('v\ \textit{literal} \times \textit{nat})\ \textit{option} \Rightarrow 'v\ \textit{twl-st-wl} \Rightarrow 'v\ \textit{twl-st-l}$› **where**
‹$\textit{st-l-of-wl None}\ (M, N, D, NE, UE, Q, W) = (M, N, D, NE, UE, \{\#\}, Q)$›
| ‹$\textit{st-l-of-wl}\ (\textit{Some}\ (L, j))\ (M, N, D, NE, UE, Q, W) =$
$(M, N, D, NE, UE, (\textit{if}\ D \neq \textit{None then}\ \{\#\}\ \textit{else clauses-to-update-wl}\ (M, N, D, NE, UE, Q, W)$
$L\ j,$
$Q))$›

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) ∈ *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* ⟹ *clauses-to-update-l T* = {#}⟩
⟨L ≠ *None* ⟹ *get-conflict-wl S* ≠ *None* ⟹ *clauses-to-update-l T* = {#}⟩
⟨L ≠ *None* ⟹ *get-conflict-wl S* = *None* ⟹ *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') ∈ *state-wl-l None* ⟹
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* = (λC (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*]:
 ‹$get\text{-}unit\text{-}init\text{-}clss\text{-}wl\ (set\text{-}conflict\text{-}wl\ D\ S) = get\text{-}unit\text{-}init\text{-}clss\text{-}wl\ S$›
 ‹$get\text{-}unit\text{-}clauses\text{-}wl\ (set\text{-}conflict\text{-}wl\ D\ S) = get\text{-}unit\text{-}clauses\text{-}wl\ S$›
 ⟨*proof*⟩

lemma *state-wl-l-mark-of-is-decided*:
 ‹$(x, y) \in state\text{-}wl\text{-}l\ b \Longrightarrow$
 $get\text{-}trail\text{-}wl\ x \neq [] \Longrightarrow$
 $is\text{-}decided\ (hd\ (get\text{-}trail\text{-}l\ y)) = is\text{-}decided\ (hd\ (get\text{-}trail\text{-}wl\ x))$›
 ⟨*proof*⟩

lemma *state-wl-l-mark-of-is-proped*:
 ‹$(x, y) \in state\text{-}wl\text{-}l\ b \Longrightarrow$
 $get\text{-}trail\text{-}wl\ x \neq [] \Longrightarrow$
 $is\text{-}proped\ (hd\ (get\text{-}trail\text{-}l\ y)) = is\text{-}proped\ (hd\ (get\text{-}trail\text{-}wl\ x))$›
 ⟨*proof*⟩

We here also update the list of watched clauses *WL*.

declare *twl-st-wl*[*simp*]

definition *unit-prop-body-wl-inv* **where**
‹$unit\text{-}prop\text{-}body\text{-}wl\text{-}inv\ T\ j\ i\ L \longleftrightarrow (i < length\ (watched\text{-}by\ T\ L) \wedge j \leq i \wedge$
 $(fst\ (watched\text{-}by\ T\ L\ !\ i) \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}wl\ T) \longrightarrow$
 $(\exists T'.\ (T, T') \in state\text{-}wl\text{-}l\ (Some\ (L, i)) \wedge j \leq i \wedge$
 $unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}inv\ L\ (fst\ (watched\text{-}by\ T\ L\ !\ i))$
 $(remove\text{-}one\text{-}lit\text{-}from\text{-}wq\ (fst\ (watched\text{-}by\ T\ L\ !\ i))\ T') \wedge$
 $L \in\#\ all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ init\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}wl\ T) + get\text{-}unit\text{-}clauses\text{-}wl\ T) \wedge$
 $correct\text{-}watching\text{-}except\ j\ i\ L\ T)))$›

lemma *unit-prop-body-wl-inv-alt-def*:
 ‹$unit\text{-}prop\text{-}body\text{-}wl\text{-}inv\ T\ j\ i\ L \longleftrightarrow (i < length\ (watched\text{-}by\ T\ L) \wedge j \leq i \wedge$
 $(fst\ (watched\text{-}by\ T\ L\ !\ i) \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}wl\ T) \longrightarrow$
 $(\exists T'.\ (T, T') \in state\text{-}wl\text{-}l\ (Some\ (L, i)) \wedge$
 $unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}inv\ L\ (fst\ (watched\text{-}by\ T\ L\ !\ i))$
 $(remove\text{-}one\text{-}lit\text{-}from\text{-}wq\ (fst\ (watched\text{-}by\ T\ L\ !\ i))\ T') \wedge$
 $L \in\#\ all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ init\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}wl\ T) + get\text{-}unit\text{-}clauses\text{-}wl\ T) \wedge$
 $correct\text{-}watching\text{-}except\ j\ i\ L\ T \wedge$
 $get\text{-}conflict\text{-}wl\ T = None \wedge$
 $length\ (get\text{-}clauses\text{-}wl\ T \propto fst\ (watched\text{-}by\ T\ L\ !\ i)) \geq 2)))$›
 (**is** ‹$?A = ?B$›)
⟨*proof*⟩

definition *propagate-lit-wl* :: ‹$'v\ literal \Rightarrow nat \Rightarrow nat \Rightarrow 'v\ twl\text{-}st\text{-}wl \Rightarrow 'v\ twl\text{-}st\text{-}wl$› **where**
 ‹$propagate\text{-}lit\text{-}wl = (\lambda L'\ C\ i\ (M, N, D, NE, UE, Q, W).$
 $let\ N = N(C \hookrightarrow swap\ (N \propto C)\ 0\ (Suc\ 0 - i))\ in$
 $(Propagated\ L'\ C\ \#\ M, N, D, NE, UE, add\text{-}mset\ (-L')\ Q, W))$›

definition *keep-watch* **where**
 ‹$keep\text{-}watch = (\lambda L\ i\ j\ (M, N, D, NE, UE, Q, W).$
 $(M, N, D, NE, UE, Q, W(L := W\ L[i := W\ L\ !\ j])))$›

lemma *length-watched-by-keep-watch*[*twl-st-wl*]:
 ‹$length\ (watched\text{-}by\ (keep\text{-}watch\ L\ i\ j\ S)\ K) = length\ (watched\text{-}by\ S\ K)$›
 ⟨*proof*⟩

lemma *watched-by-keep-watch-neq*[*twl-st-wl*, *simp*]:

$\langle w < \mathit{length}\ (\mathit{watched\text{-}by}\ S\ L) \Longrightarrow \mathit{watched\text{-}by}\ (\mathit{keep\text{-}watch}\ L\ j\ w\ S)\ L\ !\ w = \mathit{watched\text{-}by}\ S\ L\ !\ w\rangle$
⟨*proof*⟩

lemma *watched-by-keep-watch-eq*[*twl-st-wl*, *simp*]:
$\langle j < \mathit{length}\ (\mathit{watched\text{-}by}\ S\ L) \Longrightarrow \mathit{watched\text{-}by}\ (\mathit{keep\text{-}watch}\ L\ j\ w\ S)\ L\ !\ j = \mathit{watched\text{-}by}\ S\ L\ !\ w\rangle$
⟨*proof*⟩

definition *update-clause-wl* :: $\langle 'v\ \mathit{literal} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow 'v\ \mathit{twl\text{-}st\text{-}wl} \Rightarrow (\mathit{nat} \times \mathit{nat} \times 'v\ \mathit{twl\text{-}st\text{-}wl})\ \mathit{nres}\rangle$ **where**
$\langle \mathit{update\text{-}clause\text{-}wl} = (\lambda(L{::}'v\ \mathit{literal})\ C\ b\ j\ w\ i\ f\ (M, N, D, NE, UE, Q, W).\ \mathit{do}\ \{$
$\quad \mathit{let}\ K' = (N{\propto}C)\ !\ f;$
$\quad \mathit{let}\ N' = N(C \hookrightarrow \mathit{swap}\ (N \propto C)\ i\ f);$
$\quad \mathit{RETURN}\ (j, w{+}1, (M, N', D, NE, UE, Q, W(K' := W\ K'\ @\ [(C, L, b)])))$
$\})\rangle$

definition *update-blit-wl* :: $\langle 'v\ \mathit{literal} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow 'v\ \mathit{literal} \Rightarrow 'v\ \mathit{twl\text{-}st\text{-}wl} \Rightarrow (\mathit{nat} \times \mathit{nat} \times 'v\ \mathit{twl\text{-}st\text{-}wl})\ \mathit{nres}\rangle$ **where**
$\langle \mathit{update\text{-}blit\text{-}wl} = (\lambda(L{::}'v\ \mathit{literal})\ C\ b\ j\ w\ K\ (M, N, D, NE, UE, Q, W).\ \mathit{do}\ \{$
$\quad \mathit{RETURN}\ (j{+}1, w{+}1, (M, N, D, NE, UE, Q, W(L := W\ L[j{:=}(C, K, b)])))$
$\})\rangle$

definition *unit-prop-body-wl-find-unwatched-inv* **where**
$\langle \mathit{unit\text{-}prop\text{-}body\text{-}wl\text{-}find\text{-}unwatched\text{-}inv}\ f\ C\ S \longleftrightarrow$
$\quad \mathit{get\text{-}clauses\text{-}wl}\ S \propto C \neq [] \wedge$
$\quad (f = \mathit{None} \longleftrightarrow (\forall L {\in\#} \mathit{mset}\ (\mathit{unwatched\text{-}l}\ (\mathit{get\text{-}clauses\text{-}wl}\ S \propto C)).\ -\ L \in \mathit{lits\text{-}of\text{-}l}\ (\mathit{get\text{-}trail\text{-}wl}\ S)))\rangle$

abbreviation *remaining-nondom-wl* **where**
$\langle \mathit{remaining\text{-}nondom\text{-}wl}\ w\ L\ S \equiv$
$\quad (\mathit{if}\ \mathit{get\text{-}conflict\text{-}wl}\ S = \mathit{None}$
$\quad\quad \mathit{then}\ \mathit{size}\ (\mathit{filter\text{-}mset}\ (\lambda(i, \text{-}).\ i \notin\# \mathit{dom\text{-}m}\ (\mathit{get\text{-}clauses\text{-}wl}\ S))\ (\mathit{mset}\ (\mathit{drop}\ w\ (\mathit{watched\text{-}by}\ S\ L))))\ \mathit{else}\ 0)\rangle$

definition *unit-propagation-inner-loop-wl-loop-inv* **where**
$\langle \mathit{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv}\ L = (\lambda(j, w, S).$
$\quad (\exists S'.\ (S, S') \in \mathit{state\text{-}wl\text{-}l}\ (\mathit{Some}\ (L, w)) \wedge j{\le}\ w \wedge$
$\quad\quad \mathit{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv}\ L\ (S', \mathit{remaining\text{-}nondom\text{-}wl}\ w\ L\ S) \wedge$
$\quad\quad \mathit{correct\text{-}watching\text{-}except}\ j\ w\ L\ S \wedge w \le \mathit{length}\ (\mathit{watched\text{-}by}\ S\ L)))\rangle$

lemma *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*:
assumes
j-w: $\langle j \le w\rangle$ **and**
w-le: $\langle w < \mathit{length}\ (\mathit{watched\text{-}by}\ S\ L)\rangle$ **and**
corr: $\langle \mathit{correct\text{-}watching\text{-}except}\ j\ w\ L\ S\rangle$
shows $\langle \mathit{correct\text{-}watching\text{-}except}\ (\mathit{Suc}\ j)\ (\mathit{Suc}\ w)\ L\ (\mathit{keep\text{-}watch}\ L\ j\ w\ S)\rangle$
⟨*proof*⟩

lemma *correct-watching-except-update-blit*:
assumes
corr: $\langle \mathit{correct\text{-}watching\text{-}except}\ i\ j\ L\ (a, b, c, d, e, f, g(L := g\ L[j' := (x1, C, b')]))\rangle$ **and**
C': $\langle C' \in\# \mathit{all\text{-}lits\text{-}of\text{-}mm}\ (\mathit{mset}\ `\#\ \mathit{ran\text{-}mf}\ b + (d + e))\rangle$
$\langle C' \in \mathit{set}\ (b \propto x1)\rangle$
$\langle C' \neq L\rangle$

⟨*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 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 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)$\propto$$C$) ! (*1* − *i*);
let val-L' = *polarity* (*get-trail-wl* S) L';
if val-L' = *Some True*
then update-blit-wl L C b j w L' S
else do {
f ← *find-unwatched-l* (*get-trail-wl* S) (*get-clauses-wl* S $\propto$$C$);
ASSERT (*unit-prop-body-wl-find-unwatched-inv* f C S);
case f *of*
None ⇒ *do* {
if val-L' = *Some False*
then do {*RETURN* (j+*1*, w+*1*, *set-conflict-wl* (*get-clauses-wl* S $\propto$ C) S)}
else do {*RETURN* (j+*1*, w+*1*, *propagate-lit-wl* L' C i S)}
}
| *Some* f ⇒ *do* {
let K = *get-clauses-wl* S $\propto$ C ! f;
let val-L' = *polarity* (*get-trail-wl* S) K;
if val-L' = *Some True*
then update-blit-wl L C b j w K S
else update-clause-wl L C b j w i f S
}
}
}
}
}›

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')$\propto$$C$) ! *0* = L *then 0 else 1*);
let L' = ((*get-clauses-wl* S')$\propto$$C$) ! (*1* − *i*);
let val-L' = *polarity* (*get-trail-wl* S') L';
if val-L' = *Some True*

then update-blit-wl L C b j w L′ S′
else do {
f ← *find-unwatched-l* (*get-trail-wl S′*) (*get-clauses-wl S′*∝*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*⟩

1.4.3 The Functions

Inner Loop

lemma *int-xor-3-same2*: ‹*a XOR b XOR a* = *b*› **for** *a b* :: *int*
⟨*proof*⟩

lemma *nat-xor-3-same2*: ‹*a XOR b XOR a* = *b*› **for** *a b* :: *nat*
⟨*proof*⟩

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* (λ*b*. (*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* (λ*K*. *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 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 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 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 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*]: ‹$C' \equiv$ *fst* (*watched-by* S L ! w)›
defines
[*simp*]: ‹$T \equiv$ *remove-one-lit-from-wq* C' S'›

defines
[*simp*]: ‹$C'' \equiv$ *get-clauses-l* $S' \propto C'$›
assumes
S-S': ‹$(S, S') \in$ *state-wl-l* (*Some* (L, w))› **and**
w-le: ‹$w <$ *length* (*watched-by* S L)› **and**
j-w: ‹$j \leq w$› **and**
corr-w: ‹*correct-watching-except* j w L S› **and**
inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv* L (j, w, S)› **and**
n: ‹$n =$ *size* (*filter-mset* ($\lambda(i, \text{-})$. $i \notin\#$ *dom-m* (*get-clauses-wl* S)) (*mset* (*drop* w (*watched-by* S L))))›
and
confl-S: ‹*get-conflict-wl* $S =$ *None*›
shows *unit-propagation-inner-loop-body-wl-wl-int*: ‹*unit-propagation-inner-loop-body-wl* L j w $S \leq$
$\Downarrow$ *Id* (*unit-propagation-inner-loop-body-wl-int* L j w S)›
⟨*proof*⟩

lemma
fixes S :: ‹$'v$ *twl-st-wl*› **and** S' :: ‹$'v$ *twl-st-l*› **and** L :: ‹$'v$ *literal*› **and** w :: *nat*
defines [*simp*]: ‹$C' \equiv$ *fst* (*watched-by* S L ! w)›
defines
[*simp*]: ‹$T \equiv$ *remove-one-lit-from-wq* C' S'›

defines
[*simp*]: ‹$C'' \equiv$ *get-clauses-l* $S' \propto C'$›
assumes
S-S': ‹$(S, S') \in$ *state-wl-l* (*Some* (L, w))› **and**
w-le: ‹$w <$ *length* (*watched-by* S L)› **and**
j-w: ‹$j \leq w$› **and**
corr-w: ‹*correct-watching-except* j w L S› **and**
inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv* L (j, w, S)› **and**
n: ‹$n =$ *size* (*filter-mset* ($\lambda(i, \text{-})$. $i \notin\#$ *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* ($\lambda(i, \text{-})$. $i \notin\#$ *dom-m* (*get-clauses-wl* T')) (*mset* (*drop* j (*watched-by* T'

$L))))) \wedge$
$(get\text{-}conflict\text{-}wl\ T' \neq None \longrightarrow n = 0)\}$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L\ (S',\ n))$› (**is** ‹$?propa$› **is** ‹$- \leq \Downarrow\ ?unit\ -$›)**and**
unit-propagation-inner-loop-body-wl-update:
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}inv\ L\ C'\ T \Longrightarrow$
$mset\ `\#\ (ran\text{-}mf\ ((get\text{-}clauses\text{-}wl\ S)\ (C' \hookrightarrow (swap\ (get\text{-}clauses\text{-}wl\ S \propto C')\ 0$
$(1 - (if\ (get\text{-}clauses\text{-}wl\ S) \propto C'\ !\ 0 = L\ then\ 0\ else\ 1)))))) =$
$mset\ `\#\ (ran\text{-}mf\ (get\text{-}clauses\text{-}wl\ S))$› (**is** ‹$- \Longrightarrow ?eq$›)
⟨*proof*⟩

lemma
fixes S :: ‹$'v\ twl\text{-}st\text{-}wl$› **and** S' :: ‹$'v\ twl\text{-}st\text{-}l$› **and** L :: ‹$'v\ literal$› **and** w :: nat
defines [*simp*]: ‹$C' \equiv fst\ (watched\text{-}by\ S\ L\ !\ w)$›
defines
[*simp*]: ‹$T \equiv remove\text{-}one\text{-}lit\text{-}from\text{-}wq\ C'\ S'$›

defines
[*simp*]: ‹$C'' \equiv get\text{-}clauses\text{-}l\ S' \propto C'$›
assumes
S-S': ‹$(S,\ S') \in state\text{-}wl\text{-}l\ (Some\ (L,\ w))$› **and**
w-le: ‹$w < length\ (watched\text{-}by\ S\ L)$› **and**
j-w: ‹$j \leq w$› **and**
corr-w: ‹$correct\text{-}watching\text{-}except\ j\ w\ L\ S$› **and**
inner-loop-inv: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L\ (j,\ w,\ S)$› **and**
n: ‹$n = size\ (filter\text{-}mset\ (\lambda(i,\ \text{-}).\ i \notin\#\ dom\text{-}m\ (get\text{-}clauses\text{-}wl\ S))\ (mset\ (drop\ w\ (watched\text{-}by\ S\ L))))$›
and
confl-S: ‹$get\text{-}conflict\text{-}wl\ S = None$›
shows *unit-propagation-inner-loop-body-wl-spec*: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}wl\ L\ j\ w\ S \leq$
$\Downarrow\{((i,\ j,\ T'),\ (T,\ n)).$
$(T',\ T) \in state\text{-}wl\text{-}l\ (Some\ (L,\ j)) \wedge$
$correct\text{-}watching\text{-}except\ i\ j\ L\ T' \wedge$
$j \leq length\ (watched\text{-}by\ T'\ L) \wedge$
$length\ (watched\text{-}by\ S\ L) = length\ (watched\text{-}by\ T'\ L) \wedge$
$i \leq j \wedge$
$(get\text{-}conflict\text{-}wl\ T' = None \longrightarrow$
$n = size\ (filter\text{-}mset\ (\lambda(i,\ \text{-}).\ i \notin\#\ dom\text{-}m\ (get\text{-}clauses\text{-}wl\ T'))\ (mset\ (drop\ j\ (watched\text{-}by\ T'$
$L))))) \wedge$
$(get\text{-}conflict\text{-}wl\ T' \neq None \longrightarrow n = 0)\}$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L\ (S',\ n))$›
⟨*proof*⟩

definition *unit-propagation-inner-loop-wl-loop*
:: ‹$'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}wl \Rightarrow (nat \times nat \times 'v\ twl\text{-}st\text{-}wl)\ nres$› **where**
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\ L\ S_0 = do\ \{$
$let\ n = length\ (watched\text{-}by\ S_0\ L);$
$WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L}$
$(\lambda(j,\ w,\ S).\ w < n \wedge get\text{-}conflict\text{-}wl\ S = None)$
$(\lambda(j,\ w,\ S).\ do\ \{$
$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}wl\ L\ j\ w\ S$
$\})$
$(0,\ 0,\ S_0)$
$\}$›

lemma *correct-watching-except-correct-watching-cut-watch*:
  assumes *corr*: ‹*correct-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*);
    $WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L}$
      (λ(*j*, *w*, *S*). *w* < *n* ∧ *get-conflict-wl S* = *None*)
      (λ(*j*, *w*, *S*). *do* {
        unit-propagation-inner-loop-body-wl 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* ∧
      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*)
$\in \{(T'{::}\ 'v\ twl\text{-}st\text{-}wl,\ T).$
$(T',\ T) \in state\text{-}wl\text{-}l\ None\ \wedge$
$correct\text{-}watching\ T'\ \wedge$
$get\text{-}conflict\text{-}wl\ T' = None\} \rightarrow$
$\langle\{((b',\ T'),\ (b,\ T)).\ b' = b\ \wedge$
$(T',\ T) \in state\text{-}wl\text{-}l\ None\ \wedge$
$correct\text{-}watching\ T'\}\rangle nres\text{-}rel$⟩
⟨*proof*⟩

Skip or Resolve

definition *tl-state-wl* :: ⟨$'v\ twl\text{-}st\text{-}wl \Rightarrow\ 'v\ twl\text{-}st\text{-}wl$⟩ **where**
⟨$tl\text{-}state\text{-}wl = (\lambda(M,\ N,\ D,\ NE,\ UE,\ WS,\ Q).\ (tl\ M,\ N,\ D,\ NE,\ UE,\ WS,\ Q))$⟩

definition *resolve-cls-wl'* :: ⟨$'v\ twl\text{-}st\text{-}wl \Rightarrow nat \Rightarrow\ 'v\ literal \Rightarrow\ 'v\ clause$⟩ **where**
⟨$resolve\text{-}cls\text{-}wl'\ S\ C\ L\ =$
$remove1\text{-}mset\ (-L)\ (the\ (get\text{-}conflict\text{-}wl\ S) \cup\#\ (mset\ (tl\ (get\text{-}clauses\text{-}wl\ S \propto C))))$⟩

definition *update-confl-tl-wl* :: ⟨$nat \Rightarrow\ 'v\ literal \Rightarrow\ 'v\ twl\text{-}st\text{-}wl \Rightarrow bool \times\ 'v\ twl\text{-}st\text{-}wl$⟩ **where**
⟨$update\text{-}confl\text{-}tl\text{-}wl = (\lambda C\ L\ (M,\ N,\ D,\ NE,\ UE,\ WS,\ Q).$
$let\ D = resolve\text{-}cls\text{-}wl'\ (M,\ N,\ D,\ NE,\ UE,\ WS,\ Q)\ C\ L\ in$
$(False,\ (tl\ M,\ N,\ Some\ D,\ NE,\ UE,\ WS,\ Q)))$⟩

definition *skip-and-resolve-loop-wl-inv* :: ⟨$'v\ twl\text{-}st\text{-}wl \Rightarrow bool \Rightarrow\ 'v\ twl\text{-}st\text{-}wl \Rightarrow bool$⟩ **where**
⟨$skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl\text{-}inv\ S_0\ brk\ S \longleftrightarrow$
$(\exists\, S'\ S'_0.\ (S,\ S') \in state\text{-}wl\text{-}l\ None\ \wedge$
$(S_0,\ S'_0) \in state\text{-}wl\text{-}l\ None\ \wedge$
$skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\text{-}l\ S'_0\ brk\ S'\ \wedge$
$correct\text{-}watching\ S)$⟩

definition *skip-and-resolve-loop-wl* :: ⟨$'v\ twl\text{-}st\text{-}wl \Rightarrow\ 'v\ twl\text{-}st\text{-}wl\ nres$⟩ **where**
⟨$skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl\ S_0 =$
$do\ \{$
$ASSERT(get\text{-}conflict\text{-}wl\ S_0 \neq None);$
$(\text{-},\ S) \leftarrow$
$WHILE_T{}^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl\text{-}inv\ S_0\ brk\ S}$
$(\lambda(brk,\ S).\ \neg brk \wedge \neg is\text{-}decided\ (hd\ (get\text{-}trail\text{-}wl\ S)))$
$(\lambda(\text{-},\ S).$
$do\ \{$
$let\ D' = the\ (get\text{-}conflict\text{-}wl\ S);$
$let\ (L,\ C) = lit\text{-}and\text{-}ann\text{-}of\text{-}propagated\ (hd\ (get\text{-}trail\text{-}wl\ S));$
$if\ -L \notin\#\ D'\ then$
$do\ \{RETURN\ (False,\ tl\text{-}state\text{-}wl\ S)\}$
$else$
$if\ get\text{-}maximum\text{-}level\ (get\text{-}trail\text{-}wl\ S)\ (remove1\text{-}mset\ (-L)\ D') = count\text{-}decided\ (get\text{-}trail\text{-}wl$
$S)$
$then$
$do\ \{RETURN\ (update\text{-}confl\text{-}tl\text{-}wl\ C\ L\ S)\}$
$else$
$do\ \{RETURN\ (True,\ S)\}$
$\}$
$)$

$(False,\ S_0);$
$RETURN\ S$
}
›

lemma *tl-state-wl-tl-state-l*:
‹$(S,\ S') \in state\text{-}wl\text{-}l\ None \Longrightarrow (tl\text{-}state\text{-}wl\ S,\ tl\text{-}state\text{-}l\ S') \in state\text{-}wl\text{-}l\ None$›
⟨*proof*⟩

lemma *skip-and-resolve-loop-wl-spec*:
‹$(skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl,\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}l)$
$\in \{(T'::'v\ twl\text{-}st\text{-}wl,\ T).$
$(T',\ T) \in state\text{-}wl\text{-}l\ None\ \wedge$
$correct\text{-}watching\ T'\ \wedge$
$0 < count\text{-}decided\ (get\text{-}trail\text{-}wl\ T')\} \to$
$\langle\{(T',\ T).$
$(T',\ T) \in state\text{-}wl\text{-}l\ None\ \wedge$
$correct\text{-}watching\ T'\}\rangle nres\text{-}rel$›
(**is** ‹$?s \in ?A \to \langle ?B\rangle nres\text{-}rel$›)
⟨*proof*⟩

Backtrack

definition *find-decomp-wl* :: ‹$'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}wl \Rightarrow 'v\ twl\text{-}st\text{-}wl\ nres$› **where**
‹$find\text{-}decomp\text{-}wl = (\lambda L\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W).$
$SPEC(\lambda S.\ \exists K\ M2\ M1.\ S = (M1,\ N,\ D,\ NE,\ UE,\ Q,\ W) \wedge (Decided\ K\ \#\ M1,\ M2) \in set$
$(get\text{-}all\text{-}ann\text{-}decomposition\ M)\ \wedge$
$get\text{-}level\ M\ K = get\text{-}maximum\text{-}level\ M\ (the\ D - \{\#-L\#\}) + 1))$›

definition *find-lit-of-max-level-wl* :: ‹$'v\ twl\text{-}st\text{-}wl \Rightarrow 'v\ literal \Rightarrow 'v\ literal\ nres$› **where**
‹$find\text{-}lit\text{-}of\text{-}max\text{-}level\text{-}wl = (\lambda(M,\ N,\ D,\ NE,\ UE,\ Q,\ W)\ L.$
$SPEC(\lambda L'.\ L' \in\#\ remove1\text{-}mset\ (-L)\ (the\ D) \wedge get\text{-}level\ M\ L' = get\text{-}maximum\text{-}level\ M\ (the\ D -$
$\{\#-L\#\})))$›

fun *extract-shorter-conflict-wl* :: ‹$'v\ twl\text{-}st\text{-}wl \Rightarrow 'v\ twl\text{-}st\text{-}wl\ nres$› **where**
‹$extract\text{-}shorter\text{-}conflict\text{-}wl\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W) = SPEC(\lambda S.$
$\exists D'.\ D' \subseteq\#\ the\ D \wedge S = (M,\ N,\ Some\ D',\ NE,\ UE,\ Q,\ W)\ \wedge$
$clause\ `\#\ twl\text{-}clause\text{-}of\ `\#\ ran\text{-}mf\ N + NE + UE \models pm\ D' \wedge -(lit\text{-}of\ (hd\ M)) \in\#\ D')$›

declare *extract-shorter-conflict-wl.simps*[*simp del*]
lemmas *extract-shorter-conflict-wl-def = extract-shorter-conflict-wl.simps*

definition *backtrack-wl-inv* **where**
‹$backtrack\text{-}wl\text{-}inv\ S \longleftrightarrow (\exists S'.\ (S,\ S') \in state\text{-}wl\text{-}l\ None \wedge backtrack\text{-}l\text{-}inv\ S' \wedge correct\text{-}watching\ S)$
›

Rougly: we get a fresh index that has not yet been used.

definition *get-fresh-index-wl* :: ‹$'v\ clauses\text{-}l \Rightarrow \text{-} \Rightarrow \text{-} \Rightarrow nat\ nres$› **where**
‹$get\text{-}fresh\text{-}index\text{-}wl\ N\ NUE\ W = SPEC(\lambda i.\ i > 0 \wedge i \notin\#\ dom\text{-}m\ N\ \wedge$
$(\forall L \in\#\ all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ N + NUE)\ .\ i \notin fst\ `\ set\ (W\ L)))$›

definition *propagate-bt-wl* :: ‹$'v\ literal \Rightarrow 'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}wl \Rightarrow 'v\ twl\text{-}st\text{-}wl\ nres$› **where**
‹$propagate\text{-}bt\text{-}wl = (\lambda L\ L'\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W).\ do\ \{$
$D'' \leftarrow list\text{-}of\text{-}mset\ (the\ D);$

$i \leftarrow$ *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));
$S \leftarrow$ *extract-shorter-conflict-wl* S;
$S \leftarrow$ *find-decomp-wl* L S;

if size (*the* (*get-conflict-wl* S)) > *1*
then do {
$L' \leftarrow$ *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') ⟷
$M = M' \wedge N = N' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$›

fun *equality-except-trail-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*equality-except-trail-wl* (M, N, D, NE, UE, WS, Q) (M', N', D', NE', UE', WS', Q') ⟷
$N = N' \wedge D = D' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$›

lemma *equality-except-conflict-wl-get-clauses-wl*:
‹*equality-except-conflict-wl* S Y ⟹ *get-clauses-wl* S = *get-clauses-wl* Y›

⟨*proof*⟩
lemma *equality-except-trail-wl-get-clauses-wl*:
‹*equality-except-trail-wl S Y* $\Longrightarrow$ *get-clauses-wl S* = *get-clauses-wl Y*›
⟨*proof*⟩

lemma *backtrack-wl-spec*:
‹(*backtrack-wl*, *backtrack-l*)
$\in$ {(*T'*::'*v twl-st-wl*, *T*).
(*T'*, *T*) $\in$ *state-wl-l None* $\wedge$
correct-watching T' $\wedge$
get-conflict-wl T' $\neq$ *None* $\wedge$
get-conflict-wl T' $\neq$ *Some* {#}} $\rightarrow$
⟨{(*T'*, *T*).
(*T'*, *T*) $\in$ *state-wl-l None* $\wedge$
correct-watching T'}⟩*nres-rel*›
(**is** ‹*?bt* $\in$ *?A* $\rightarrow$ ⟨*?B*⟩*nres-rel*›)
⟨*proof*⟩

Backtrack, Skip, Resolve or Decide

definition *cdcl-twl-o-prog-wl-pre* **where**
‹*cdcl-twl-o-prog-wl-pre S* $\longleftrightarrow$
($\exists$ *S'*. (*S*, *S'*) $\in$ *state-wl-l None* $\wedge$
correct-watching S $\wedge$
cdcl-twl-o-prog-l-pre S')›

definition *cdcl-twl-o-prog-wl* :: ‹'*v twl-st-wl* $\Rightarrow$ (*bool* $\times$ '*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* $\neq$ *None* $\wedge$ *get-conflict-wl T* $\neq$ *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*) $\in$ {(*S*::'*v twl-st-wl*, *S'*::'*v twl-st-l*).
(*S*, *S'*) $\in$ *state-wl-l None* $\wedge$
correct-watching S} $\rightarrow_f$
⟨{((*brk*::*bool*, *T*::'*v twl-st-wl*), *brk'*::*bool*, *T'*::'*v twl-st-l*).
(*T*, *T'*) $\in$ *state-wl-l None* $\wedge$
brk = *brk'* $\wedge$
correct-watching T}⟩*nres-rel*›

(**is** ‹$?o \in ?A \rightarrow_f \langle ?B \rangle$ *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* $S_0 \equiv \lambda(brk,\ T).$
$(\exists\ T'\ S_0'.\ (T,\ T') \in$ *state-wl-l None* ∧
$(S_0,\ S_0') \in$ *state-wl-l None* ∧
cdcl-twl-stgy-prog-l-inv S_0' $(brk,\ T'))$›

definition *cdcl-twl-stgy-prog-wl* :: ‹$'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl nres*› **where**
‹*cdcl-twl-stgy-prog-wl* S_0 =
do {
$(brk,\ T) \leftarrow$ $WHILE_T{}^{\textit{cdcl-twl-stgy-prog-wl-inv}\ S_0}$
$(\lambda(brk,\ \text{-}).\ \neg brk)$
$(\lambda(brk,\ S).$ *do* {
$T \leftarrow$ *unit-propagation-outer-loop-wl* S;
cdcl-twl-o-prog-wl T
})
$(False,\ S_0)$;
RETURN T
}›

theorem *cdcl-twl-stgy-prog-wl-spec*:
‹(*cdcl-twl-stgy-prog-wl*, *cdcl-twl-stgy-prog-l*) ∈ {(S::$'v$ *twl-st-wl*, S').
$(S,\ S') \in$ *state-wl-l None* ∧
correct-watching S} →
⟨*state-wl-l None*⟩*nres-rel*›
(**is** ‹$?o \in ?A \rightarrow \langle ?B \rangle$ *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').
$(S,\ S') \in$ *state-wl-l None* ∧ *correct-watching* S} →
⟨{(S::$'v$ *twl-st-wl*, S').
$(S,\ S') \in$ *state-wl-l None* ∧ *correct-watching* S}⟩*nres-rel*›
(**is** ‹$?o \in ?A \rightarrow \langle ?B \rangle$ *nres-rel*›)
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-wl-pre* **where**
‹*cdcl-twl-stgy-prog-wl-pre* S U ⟷
(∃ T. $(S,\ T) \in$ *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* $S \le \Downarrow$ (*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 {

$b \leftarrow SPEC(\lambda\text{-}.\ True)$;
$(b,\ brk,\ T) \leftarrow WHILE_T{}^{\lambda(\text{-},\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}inv\ S_0\ S}$
$(\lambda(b,\ brk,\ \text{-}).\ b \wedge \neg brk)$
$(\lambda(\text{-},\ brk,\ S).\ do\ \{$
$T \leftarrow unit\text{-}propagation\text{-}outer\text{-}loop\text{-}wl\ S$;
$T \leftarrow cdcl\text{-}twl\text{-}o\text{-}prog\text{-}wl\ T$;
$b \leftarrow SPEC(\lambda\text{-}.\ True)$;
$RETURN\ (b,\ T)$
$\})$
$(b,\ False,\ S_0)$;
if brk then RETURN T
else cdcl-twl-stgy-prog-wl T
$\}$›

theorem *cdcl-twl-stgy-prog-break-wl-spec′*:
‹$(cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}break\text{-}wl,\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}break\text{-}l) \in \{(S::'v\ twl\text{-}st\text{-}wl,\ S').$
$(S,\ S') \in state\text{-}wl\text{-}l\ None \wedge correct\text{-}watching\ S\} \rightarrow_f$
$\langle\{(S::'v\ twl\text{-}st\text{-}wl,\ S').\ (S,\ S') \in state\text{-}wl\text{-}l\ None \wedge correct\text{-}watching\ S\}\rangle nres\text{-}rel$›
(**is** ‹$?o \in\ ?A \rightarrow_f \langle ?B\rangle\ nres\text{-}rel$›)
⟨*proof*⟩

theorem *cdcl-twl-stgy-prog-break-wl-spec*:
‹$(cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}break\text{-}wl,\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}break\text{-}l) \in \{(S::'v\ twl\text{-}st\text{-}wl,\ S').$
$(S,\ S') \in state\text{-}wl\text{-}l\ None \wedge$
$correct\text{-}watching\ S\} \rightarrow_f$
$\langle state\text{-}wl\text{-}l\ None\rangle nres\text{-}rel$›
(**is** ‹$?o \in\ ?A \rightarrow_f \langle ?B\rangle\ nres\text{-}rel$›)
⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-break-wl-spec-final*:
assumes
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}pre\ S\ S'$›
shows
‹$cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}break\text{-}wl\ S \le \Downarrow (state\text{-}wl\text{-}l\ None\ O\ twl\text{-}st\text{-}l\ None)\ (conclusive\text{-}TWL\text{-}run\ S')$›
⟨*proof*⟩

end
theory *Watched-Literals-Watch-List-Domain*
imports *Watched-Literals-Watch-List*
Array-UInt
begin

We refine the implementation by adding a *domain* on the literals

no-notation *Ref.update* (- := - *62*)

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*›
type-synonym *lit-queue-l* = ‹*uint32 list*›
type-synonym *nat-trail* = ‹(*uint32* × *nat option*) *list*›
type-synonym *clause-wl* = ‹*uint32 array*›
type-synonym *unit-lits-wl* = ‹*uint32 list list*›

1.4.5 Refinement

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).

locale *isasat-input-ops* =
  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 *blits-in-*$\mathcal{L}_{in}$ :: ‹*nat twl-st-wl* ⇒ *bool*› **where**
  ‹*blits-in-*$\mathcal{L}_{in}$ *S* ⟷
    (∀ *L* ∈# $\mathcal{L}_{all}$. ∀ (*i*, *K*, *b*) ∈ *set* (*watched-by S L*). *K* ∈# $\mathcal{L}_{all}$)›

definition *literals-are-*$\mathcal{L}_{in}$ :: ‹*nat twl-st-wl* ⇒ *bool*› **where**
  ‹*literals-are-*$\mathcal{L}_{in}$ *S* ≡
     is-$\mathcal{L}_{all}$ (*all-lits-of-mm* ((λ*C*. *mset* (*fst C*)) ‘# *ran-m* (*get-clauses-wl S*)
        + *get-unit-clauses-wl S*)) ∧
     blits-in-$\mathcal{L}_{in}$ *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* ∈# $\mathcal{L}_{all}$ ⟷ *atm-of x* ∈ *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*) ⟷ *literals-are-in-*$\mathcal{L}_{in}$ *A* ∧ *L* ∈# $\mathcal{L}_{all}$›
  ⟨*proof*⟩

lemma *literals-are-in-*$\mathcal{L}_{in}$*-mono*:
  assumes *N*: ‹*literals-are-in-*$\mathcal{L}_{in}$ *D'*› **and** *D*: ‹*D* ⊆# *D'*›
  shows ‹*literals-are-in-*$\mathcal{L}_{in}$ *D*›
⟨*proof*⟩

lemma *literals-are-in-*$\mathcal{L}_{in}$*-sub*:
  ‹*literals-are-in-*$\mathcal{L}_{in}$ *y* ⟹ *literals-are-in-*$\mathcal{L}_{in}$ (*y* − *z*)›
  ⟨*proof*⟩

lemma *all-lits-of-m-subset-all-lits-of-mmD*:
  ‹*a* ∈# *b* ⟹ *set-mset* (*all-lits-of-m a*) ⊆ *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*)›

$\langle proof \rangle$

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*›
$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-nth*:
fixes *C* :: *nat*
assumes *dom*: ‹*C* ∈# *dom-m* (*get-clauses-wl S*)› **and**
‹*literals-are-$\mathcal{L}_{in}$ S*›
shows ‹*literals-are-in-$\mathcal{L}_{in}$* (*mset* (*get-clauses-wl S* ∝ *C*))›
$\langle proof \rangle$

lemma *uminus-$\mathcal{A}_{in}$-iff*: ‹− *L* ∈# $\mathcal{L}_{all}$ ⟷ *L* ∈# $\mathcal{L}_{all}$›
$\langle proof \rangle$

definition *literals-are-in-$\mathcal{L}_{in}$-mm* :: ‹*nat clauses* ⇒ *bool*› **where**
‹*literals-are-in-$\mathcal{L}_{in}$-mm C* ⟷ *set-mset* (*all-lits-of-mm C*) ⊆ *set-mset* $\mathcal{L}_{all}$›

lemma *literals-are-in-$\mathcal{L}_{in}$-mm-in-$\mathcal{L}_{all}$*:
assumes
N1: ‹*literals-are-in-$\mathcal{L}_{in}$-mm* (*mset* ‘# *ran-mf xs*)› **and**
i-xs: ‹*i* ∈# *dom-m xs*› **and** *j-xs*: ‹*j* < *length* (*xs* ∝ *i*)›
shows ‹*xs* ∝ *i* ! *j* ∈# $\mathcal{L}_{all}$›
$\langle proof \rangle$

definition *literals-are-in-$\mathcal{L}_{in}$-trail* :: ‹(*nat*, ′*mark*) *ann-lits* ⇒ *bool*› **where**
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* ⟷ *set-mset* (*lit-of* ‘# *mset M*) ⊆ *set-mset* $\mathcal{L}_{all}$›

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* ⟹ *a* ∈ *lits-of-l M* ⟹ *a* ∈# $\mathcal{L}_{all}$›
$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l-atms*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* ⟹ *a* ∈ *lits-of-l M* ⟹ *atm-of a* ∈# $\mathcal{A}_{in}$›
$\langle proof \rangle$

lemma (**in** *isasat-input-ops*) *literals-are-in-$\mathcal{L}_{in}$-trail-Cons*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* (*L* # *M*) ⟷
literals-are-in-$\mathcal{L}_{in}$-trail M ∧ *lit-of L* ∈# $\mathcal{L}_{all}$›
$\langle proof \rangle$

lemma (**in** *isasat-input-ops*) *literals-are-in-$\mathcal{L}_{in}$-trail-empty*[*simp*]:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* []›
$\langle proof \rangle$

lemma (**in** *isasat-input-ops*) *literals-are-in-$\mathcal{L}_{in}$-Cons*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* (*a* # *M*) ⟷ *lit-of a* ∈# $\mathcal{L}_{all}$ ∧ *literals-are-in-$\mathcal{L}_{in}$-trail M*›
$\langle proof \rangle$

lemma (**in** *isasat-input-ops*) *literals-are-in-$\mathcal{L}_{in}$-trail-lit-of-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* = *literals-are-in-$\mathcal{L}_{in}$* (*lit-of* ‘# *mset M*)›
$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-in-mset-$\mathcal{L}_{all}$*:
‹*literals-are-in-$\mathcal{L}_{in}$ C* ⟹ *L* ∈# *C* ⟹ *L* ∈# $\mathcal{L}_{all}$›

$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-in-$\mathcal{L}_{all}$*:
assumes
N1: ‹*literals-are-in-$\mathcal{L}_{in}$ (mset xs)*› **and**
i-xs: ‹$i <$ *length xs*›
shows ‹*xs* ! $i \in\#\ \mathcal{L}_{all}$›
$\langle proof \rangle$

lemma *in-literals-are-in-$\mathcal{L}_{in}$-in-D_0*:
assumes ‹*literals-are-in-$\mathcal{L}_{in}$ D*› **and** ‹$L \in\# D$›
shows ‹$L \in\#\ \mathcal{L}_{all}$›
$\langle proof \rangle$

lemma *is-$\mathcal{L}_{all}$-alt-def*: ‹*is-$\mathcal{L}_{all}$ (all-lits-of-mm A)* $\longleftrightarrow$ *atms-of $\mathcal{L}_{all}$* = *atms-of-mm A*›
$\langle proof \rangle$

lemma *in-$\mathcal{L}_{all}$-atm-of-$\mathcal{A}_{in}$*:‹$L \in\#\ \mathcal{L}_{all} \longleftrightarrow$ *atm-of L* $\in\#\ \mathcal{A}_{in}$›
$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-alt-def*:
‹*literals-are-in-$\mathcal{L}_{in}$ S* $\longleftrightarrow$ *atms-of S* $\subseteq$ *atms-of $\mathcal{L}_{all}$*›
$\langle proof \rangle$

lemma (**in** *isasat-input-ops*)
assumes
x2-T: ‹(*x2*, *T*) $\in$ *state-wl-l b*› **and**
struct: ‹*twl-struct-invs U*› **and**
T-U: ‹(*T*, *U*) $\in$ *twl-st-l b'*›
shows
literals-are-$\mathcal{L}_{in}$-literals-are-$\mathcal{L}_{in}$-trail:
‹*literals-are-$\mathcal{L}_{in}$ x2* $\Longrightarrow$ *literals-are-in-$\mathcal{L}_{in}$-trail (get-trail-wl x2)*›
(**is** ‹- $\Longrightarrow$ *?trail*›) **and**
literals-are-$\mathcal{L}_{in}$-literals-are-in-$\mathcal{L}_{in}$-conflict:
‹*literals-are-$\mathcal{L}_{in}$ x2* $\Longrightarrow$ *get-conflict-wl x2* $\neq$ *None* $\Longrightarrow$ *literals-are-in-$\mathcal{L}_{in}$ (the (get-conflict-wl x2))*›
and
conflict-not-tautology:
‹*get-conflict-wl x2* $\neq$ *None* $\Longrightarrow \neg$*tautology (the (get-conflict-wl x2))*›
$\langle proof \rangle$

lemma (**in** *isasat-input-ops*) *literals-are-in-$\mathcal{L}_{in}$-trail-atm-of*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* $\longleftrightarrow$ *atm-of ‘ lits-of-l M* $\subseteq$ *set-mset $\mathcal{A}_{in}$*›
$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-poss-remdups-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$ (poss (remdups-mset (atm-of ‘# C)))* $\longleftrightarrow$ *literals-are-in-$\mathcal{L}_{in}$ C*›
$\langle proof \rangle$

lemma *literals-are-in-$\mathcal{L}_{in}$-negs-remdups-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$ (negs (remdups-mset (atm-of ‘# C)))* $\longleftrightarrow$ *literals-are-in-$\mathcal{L}_{in}$ C*›
$\langle proof \rangle$

end

context *isasat-input-ops*
begin

definition (**in** *isasat-input-ops*) *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-*$\mathcal{L}_{in}$ T' ∧ $L \in\# \mathcal{L}_{all}$›

- 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 (**in** −) *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 (**in** *isasat-input-ops*) *unit-propagation-inner-loop-wl-loop-D-inv* **where**
‹*unit-propagation-inner-loop-wl-loop-D-inv L* = (λ(*j*, *w*, *S*).
literals-are-$\mathcal{L}_{in}$ *S* ∧ $L \in\# \mathcal{L}_{all}$ ∧
unit-propagation-inner-loop-wl-loop-inv L (*j*, *w*, *S*))›

definition (**in** *isasat-input-ops*) *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 (**in** *isasat-input-ops*) *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 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 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
}
}
}
}
})›

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** *isasat-input-ops*) *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 (**in** *isasat-input-ops*) *literals-are-*$\mathcal{L}_{in}$*-keep-watch*[*twl-st-wl*, *simp*, *intro!*]:
‹*literals-are-*$\mathcal{L}_{in}$ *S ⟹ w < length (watched-by S K) ⟹ literals-are-*$\mathcal{L}_{in}$ *(keep-watch K j w S)*›
⟨*proof*⟩

lemma *blits-in-*$\mathcal{L}_{in}$*-propagate*:
‹*blits-in-*$\mathcal{L}_{in}$ *(Propagated A x1′ # x1b, x1aa*
(x1 ↪ swap (x1aa ∝ x1) 0 (Suc 0)), D, x1c, x1d,

add-mset A' *x1e*, *x2e*) $\longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)›
‹*blits-in-*$\mathcal{L}_{in}$ (*x1b*, *x1aa*
(*x1* $\hookrightarrow$ *swap* (*x1aa* $\propto$ *x1*) *0* (*Suc 0*)), *D*, *x1c*, *x1d*,*x1e*, *x2e*) $\longleftrightarrow$
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*)›
‹$K \in\# \mathcal{L}_{all} \Longrightarrow$ *blits-in-*$\mathcal{L}_{in}$
(*x1a*, *x1aa*(*x1'* $\hookrightarrow$ *swap* (*x1aa* $\propto$ *x1'*) *n* n'), *D*, *x1c*, *x1d*,
x1e, *x2e*
(*x1aa* $\propto$ *x1'* ! n' :=
x2e (*x1aa* $\propto$ *x1'* ! n') @ [(*x1'*, *K*, b')])) $\longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*,
x1e, *x2e*)›
⟨*proof*⟩

lemma *literals-are-*$\mathcal{L}_{in}$*-set-conflict-wl*:
‹*literals-are-*$\mathcal{L}_{in}$ (*set-conflict-wl D S*) $\longleftrightarrow$ *literals-are-*$\mathcal{L}_{in}$ *S*›
⟨*proof*⟩

lemma (**in** *isasat-input-ops*) *blits-in-*$\mathcal{L}_{in}$*-keep-watch'*:
assumes K': ‹$K' \in\# \mathcal{L}_{all}$ › **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 *unit-propagation-inner-loop-body-wl-D-spec*:
fixes *S* :: ‹*nat twl-st-wl*› **and** *K* :: ‹*nat literal*› **and** *w* :: *nat*
assumes
K: ‹$K \in\# \mathcal{L}_{all}$› **and**
$\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ *S*›
shows ‹*unit-propagation-inner-loop-body-wl-D K j w S* $\le$
$\Downarrow$ {((j', n', T'), (*j*, *n*, *T*)). $j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ T'}
(*unit-propagation-inner-loop-body-wl K j w S*)›
⟨*proof*⟩

lemma
shows *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*) $\in$
[λ(((*K*, *j*), *w*), *S*). *literals-are-*$\mathcal{L}_{in}$ *S* $\wedge$ $K \in\# \mathcal{L}_{all}$]$_f$
Id $\times_r$ *Id* $\times_r$ *Id* $\times_r$ *Id* $\to$ ⟨*nat-rel* $\times_r$ *nat-rel* $\times_r$ {(T', *T*). $T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ *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*) $\in$
[λ(((*K*, *j*), *w*), *S*). *literals-are-*$\mathcal{L}_{in}$ *S* $\wedge$ $K \in\# \mathcal{L}_{all}$]$_f$
Id $\times_r$ *Id* $\times_r$ *Id* $\times_r$ *Id* $\to$ ⟨*nat-rel* $\times_r$ *nat-rel* $\times_r$ *Id*⟩ *nres-rel*›
(**is** ‹*?G2*›)
⟨*proof*⟩

definition (**in** *isasat-input-ops*) *unit-propagation-inner-loop-wl-loop-D*
:: ‹*nat literal* $\Rightarrow$ *nat twl-st-wl* $\Rightarrow$ (*nat* $\times$ *nat* $\times$ *nat twl-st-wl*) *nres*›
where
‹*unit-propagation-inner-loop-wl-loop-D L* S_0 = *do* {

$ASSERT(L \in\# \mathcal{L}_{all})$;
let n = length (watched-by S_0 L);
$WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}D\text{-}inv\ L}$
($\lambda(j, w, S)$. $w < n \wedge$ get-conflict-wl S = None)
($\lambda(j, w, S)$. 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-$\mathcal{L}_{in}$ S› **and** K: ‹$K \in\# \mathcal{L}_{all}$›
shows ‹unit-propagation-inner-loop-wl-loop-D K S $\leq$
$\Downarrow$ {((j', n', T'), j, n, T). $j' = j \wedge n' = n \wedge T = T' \wedge$ literals-are-$\mathcal{L}_{in}$ T'}
(unit-propagation-inner-loop-wl-loop K S)›
⟨*proof*⟩

definition (**in** *isasat-input-ops*) *unit-propagation-inner-loop-wl-D*
:: ‹nat literal $\Rightarrow$ nat twl-st-wl $\Rightarrow$ nat twl-st-wl nres› **where**
‹unit-propagation-inner-loop-wl-D L S_0 = do {
(j, w, S) $\leftarrow$ unit-propagation-inner-loop-wl-loop-D L S_0;
ASSERT ($j \leq w \wedge w \leq$ length (watched-by S L) $\wedge L \in\# \mathcal{L}_{all}$);
$S \leftarrow$ cut-watch-list j w L S;
RETURN S
}›

lemma *unit-propagation-inner-loop-wl-D-spec*:
assumes $\mathcal{A}_{in}$: ‹literals-are-$\mathcal{L}_{in}$ S› **and** K: ‹$K \in\# \mathcal{L}_{all}$›
shows ‹unit-propagation-inner-loop-wl-D K S $\leq$
$\Downarrow$ {(T', T). $T = T' \wedge$ literals-are-$\mathcal{L}_{in}$ T}
(unit-propagation-inner-loop-wl K S)›
⟨*proof*⟩

definition (**in** *isasat-input-ops*) *unit-propagation-outer-loop-wl-D-inv* **where**
‹unit-propagation-outer-loop-wl-D-inv S $\longleftrightarrow$
unit-propagation-outer-loop-wl-inv S $\wedge$
literals-are-$\mathcal{L}_{in}$ S›

definition (**in** *isasat-input-ops*) *unit-propagation-outer-loop-wl-D*
:: ‹nat twl-st-wl $\Rightarrow$ nat twl-st-wl nres›
where
‹unit-propagation-outer-loop-wl-D S_0 =
$WHILE_T{}^{unit\text{-}propagation\text{-}outer\text{-}loop\text{-}wl\text{-}D\text{-}inv}$
(λS. literals-to-update-wl $S \neq$ {#})
(λS. do {
ASSERT(literals-to-update-wl $S \neq$ {#});
(S', L) $\leftarrow$ select-and-remove-from-literals-to-update-wl S;
ASSERT($L \in\#$ all-lits-of-mm (mset '# ran-mf (get-clauses-wl S') +
get-unit-clauses-wl 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-$\mathcal{L}_{in}$ (set-literals-to-update-wl C S) $\longleftrightarrow$ literals-are-$\mathcal{L}_{in}$ S›

⟨*proof*⟩

lemma *unit-propagation-outer-loop-wl-D-spec*:
assumes $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ *S*›
shows ‹*unit-propagation-outer-loop-wl-D S* $\leq$
$\Downarrow \{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $T\}$
(*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*) $\in \{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $T\} \rightarrow_f$
$\langle\{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $T\}\rangle$*nres-rel*›
⟨*proof*⟩

definition (**in** *isasat-input-ops*) *skip-and-resolve-loop-wl-D-inv* **where**
‹*skip-and-resolve-loop-wl-D-inv* S_0 *brk S* $\equiv$
skip-and-resolve-loop-wl-inv S_0 *brk S* $\wedge$ *literals-are-*$\mathcal{L}_{in}$ *S*›

definition (**in** *isasat-input-ops*) *skip-and-resolve-loop-wl-D*
:: ‹*nat twl-st-wl* $\Rightarrow$ *nat twl-st-wl nres*›
where
‹*skip-and-resolve-loop-wl-D* S_0 =
do {
ASSERT(*get-conflict-wl* $S_0 \neq$ *None*);
(-, *S*) $\leftarrow$
$WHILE_T{}^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ S}$
(λ(*brk*, *S*). $\neg$*brk* $\wedge$ $\neg$*is-decided* (*hd* (*get-trail-wl S*)))
(λ(*brk*, *S*).
do {
ASSERT($\neg$*brk* $\wedge$ $\neg$*is-decided* (*hd* (*get-trail-wl S*)));
let D' = *the* (*get-conflict-wl S*);
let (*L*, *C*) = *lit-and-ann-of-propagated* (*hd* (*get-trail-wl S*));
if $-L \notin\# D'$ *then*
do {*RETURN* (*False*, *tl-state-wl S*)}
else
if get-maximum-level (*get-trail-wl S*) (*remove1-mset* $(-L)$ D') =
count-decided (*get-trail-wl S*)
then
do {*RETURN* (*update-confl-tl-wl C L S*)}
else
do {*RETURN* (*True*, *S*)}
}
)
(*False*, S_0);
RETURN S
}
›

lemma (**in** *isasat-input-ops*) *literals-are-*$\mathcal{L}_{in}$*-tl-state-wl*[*simp*]:
‹*literals-are-*$\mathcal{L}_{in}$ (*tl-state-wl S*) = *literals-are-*$\mathcal{L}_{in}$ *S*›
⟨*proof*⟩

lemma *get-clauses-wl-tl-state*: ‹*get-clauses-wl* (*tl-state-wl T*) = *get-clauses-wl T*›
⟨*proof*⟩

lemma *skip-and-resolve-loop-wl-D-spec*:
  assumes $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ *S*›
  shows ‹*skip-and-resolve-loop-wl-D S* ≤
    ⇓ {(*T'*, *T*). *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ *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 (**in** −) *single-of-mset* **where**
  ‹*single-of-mset D* = *SPEC*(λ*L*. *D* = *mset* [*L*])›

definition (**in** *isasat-input-ops*) *backtrack-wl-D-inv* **where**
  ‹*backtrack-wl-D-inv S* ⟷ *backtrack-wl-inv S* ∧ *literals-are-*$\mathcal{L}_{in}$ *S*›

definition (**in** *isasat-input-ops*) *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 (**in** *isasat-input-ops*) *propagate-unit-bt-wl-D*
  :: ‹*nat literal* ⇒ *nat twl-st-wl* ⇒ (*nat twl-st-wl*) *nres*›
where
  ‹*propagate-unit-bt-wl-D* = (λ*L* (*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 (**in** *isasat-input-ops*) *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-*$\mathcal{L}_{in}$ S› **and** *confl*: ‹*get-conflict-wl* S ~= *None*›
  shows ‹*backtrack-wl-D* S ≤
    ⇓ $\{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $T\}$
      (*backtrack-wl* S)›
⟨*proof*⟩

Decide or Skip

thm *find-unassigned-lit-wl-def*
definition (**in** *isasat-input-ops*) *find-unassigned-lit-wl-D*
  :: ‹*nat twl-st-wl* ⇒ (*nat twl-st-wl* × *nat literal option*) *nres*›
where
  ‹*find-unassigned-lit-wl-D* S = (
     SPEC(λ((*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*), *L*).
         S = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) ∧
         (*L* ≠ *None* ⟶
            undefined-lit M (*the L*) ∧ *the L* ∈# $\mathcal{L}_{all}$ ∧
            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 (**in** *isasat-input-ops*) *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-*$\mathcal{L}_{in}$ S›

definition(**in** *isasat-input-ops*) *decide-wl-or-skip-D*
  :: ‹*nat twl-st-wl* ⇒ (*bool* × *nat twl-st-wl*) *nres*›
where
  ‹*decide-wl-or-skip-D* S = (*do* {
    ASSERT(*decide-wl-or-skip-D-pre* S);
    (S, *L*) ← *find-unassigned-lit-wl-D* S;
    case L of
      None ⇒ *RETURN* (*True*, S)
    | *Some L* ⇒ *RETURN* (*False*, *decide-lit-wl L S*)
  })
›

theorem *decide-wl-or-skip-D-spec*:
  assumes ‹*literals-are-*$\mathcal{L}_{in}$ S›
  shows ‹*decide-wl-or-skip-D* S
    ≤ ⇓ $\{((b', T'), b, T).\ b = b' \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $T\}$ (*decide-wl-or-skip* S)›
⟨*proof*⟩

Backtrack, Skip, Resolve or Decide

definition (**in** *isasat-input-ops*) *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-*$\mathcal{L}_{in}$ *S*›

definition (**in** *isasat-input-ops*) *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-*$\mathcal{L}_{in}$ *S*›
shows ‹*cdcl-twl-o-prog-wl-D S* $\leq \Downarrow$ {((*b'*, *T'*), (*b*, *T*)). *b* = *b'* $\wedge$ *T* = *T'* $\wedge$ *literals-are-*$\mathcal{L}_{in}$ *T*}
(*cdcl-twl-o-prog-wl S*)›
⟨*proof*⟩

theorem *cdcl-twl-o-prog-wl-D-spec'*:
shows
‹(*cdcl-twl-o-prog-wl-D*, *cdcl-twl-o-prog-wl*) $\in$
{(*S*,*S'*). (*S*,*S'*) $\in$ *Id* $\wedge$*literals-are-*$\mathcal{L}_{in}$ *S*} $\rightarrow_f$
⟨*bool-rel* $\times_r$ {(*T'*, *T*). *T* = *T'* $\wedge$ *literals-are-*$\mathcal{L}_{in}$ *T*}⟩ *nres-rel*›
⟨*proof*⟩

Full Strategy

definition (**in** *isasat-input-ops*) *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*) $\leftarrow$ $WHILE_T{}^{\lambda(brk,\ T).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}inv\ S_0\ (brk,\ T) \wedge \quad literals\text{-}are\text{-}\mathcal{L}_{in}\ T}$
(λ(*brk*, -). $\neg$*brk*)
(λ(*brk*, *S*).
do {
T $\leftarrow$ *unit-propagation-outer-loop-wl-D S*;
cdcl-twl-o-prog-wl-D T
})
(*False*, S_0);
RETURN T
}
}
›

theorem *cdcl-twl-stgy-prog-wl-D-spec*:
assumes ‹*literals-are-$\mathcal{L}_{in}$ S*›
shows ‹*cdcl-twl-stgy-prog-wl-D S* $\leq \Downarrow$ {(T', T). $T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$ 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-$\mathcal{L}_{in}$ S*} $\rightarrow_f$
⟨{(T', T). $T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$ T*}⟩ *nres-rel*›
⟨*proof*⟩

definition (**in** *isasat-input-ops*) *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-$\mathcal{L}_{in}$ S*)›

lemma *cdcl-twl-stgy-prog-wl-D-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-D-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-wl-D S* $\leq \Downarrow$ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S'*)›
⟨*proof*⟩

definition (**in** *isasat-input-ops*) *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\text{-}are\text{-}\mathcal{L}_{in}\ 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-$\mathcal{L}_{in}$ S*›
shows ‹*cdcl-twl-stgy-prog-break-wl-D S* $\leq \Downarrow$ {(T', T). $T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$ 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

$\langle$*cdcl-twl-stgy-prog-break-wl-D S* $\leq \Downarrow$ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S'*)$\rangle$
$\langle$*proof*$\rangle$

end — end of locale *isasat-input-ops*

The definition is here to be shared later.

definition *get-propagation-reason* :: $\langle$('*v*, '*mark*) *ann-lits* $\Rightarrow$ '*v literal* $\Rightarrow$ '*mark option nres*$\rangle$ **where**
$\langle$*get-propagation-reason M L* = *SPEC*(λC. $C \neq$ *None* $\longrightarrow$ *Propagated L* (*the C*) $\in$ *set M*)$\rangle$

end
theory *Watched-Literals-Initialisation*
imports *Watched-Literals-List*
begin

1.4.6 Initialise Data structure

type-synonym '*v twl-st-init* = $\langle$'*v twl-st* $\times$ '*v clauses*$\rangle$

fun *get-trail-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ ('*v*, '*v clause*) *ann-lit list*$\rangle$ **where**
$\langle$*get-trail-init* ((*M*, -, -, -, -, -, -, -), -) = *M*$\rangle$

fun *get-conflict-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v cconflict*$\rangle$ **where**
$\langle$*get-conflict-init* ((-, -, -, *D*, -, -, -, -), -) = *D*$\rangle$

fun *literals-to-update-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v clause*$\rangle$ **where**
$\langle$*literals-to-update-init* ((-, -, -, -, -, -, -, *Q*), -) = *Q*$\rangle$

fun *get-init-clauses-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v twl-cls multiset*$\rangle$ **where**
$\langle$*get-init-clauses-init* ((-, *N*, -, -, -, -, -, -), -) = *N*$\rangle$

fun *get-learned-clauses-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v twl-cls multiset*$\rangle$ **where**
$\langle$*get-learned-clauses-init* ((-, -, *U*, -, -, -, -, -), -) = *U*$\rangle$

fun *get-unit-init-clauses-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v clauses*$\rangle$ **where**
$\langle$*get-unit-init-clauses-init* ((-, -, -, -, *NE*, -, -, -), -) = *NE*$\rangle$

fun *get-unit-learned-clauses-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v clauses*$\rangle$ **where**
$\langle$*get-unit-learned-clauses-init* ((-, -, -, -, -, *UE*, -, -), -) = *UE*$\rangle$

fun *clauses-to-update-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ ('*v literal* $\times$ '*v twl-cls*) *multiset*$\rangle$ **where**
$\langle$*clauses-to-update-init* ((-, -, -, -, -, -, *WS*, -), -) = *WS*$\rangle$

fun *other-clauses-init* :: $\langle$'*v twl-st-init* $\Rightarrow$ '*v clauses*$\rangle$ **where**
$\langle$*other-clauses-init* ((-, -, -, -, -, -, -), *OC*) = *OC*$\rangle$

fun *add-to-init-clauses* :: $\langle$'*v clause-l* $\Rightarrow$ '*v twl-st-init* $\Rightarrow$ '*v twl-st-init*$\rangle$ **where**
$\langle$*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*)$\rangle$

fun *add-to-unit-init-clauses* :: $\langle$'*v clause* $\Rightarrow$ '*v twl-st-init* $\Rightarrow$ '*v twl-st-init*$\rangle$ **where**
$\langle$*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*)$\rangle$

fun *set-conflict-init* :: $\langle$'*v clause-l* $\Rightarrow$ '*v twl-st-init* $\Rightarrow$ '*v twl-st-init*$\rangle$ **where**
$\langle$*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* $S \neq$ *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* ⟹ $C \neq []$ ⟹ *hd* $C \in$ *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 *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: ⟨$C \neq []$⟩ **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* ⟷

$(\exists S'\ T'.\ (S,\ S') \in \textit{state-wl-l-init} \wedge (T,\ T') \in \textit{state-wl-l-init} \wedge$
$\textit{init-dt-spec}\ C\ S'\ T')$›

lemma *init-dt-wl-init-dt-wl-spec*:
assumes ‹*init-dt-wl-pre CS S*›
shows ‹$\textit{init-dt-wl}\ CS\ S \leq \textit{SPEC}\ (\textit{init-dt-wl-spec}\ CS\ S)$›
⟨*proof*⟩

fun *correct-watching-init* :: ‹'v twl-st-wl ⇒ bool› **where**
[*simp del*]: ‹$\textit{correct-watching-init}\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W) \longleftrightarrow$
$\textit{all-blits-are-in-problem-init}\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W) \wedge$
$(\forall L.$
$(\forall (i,\ K,\ b){\in}\#\textit{mset}\ (W\ L).\ i \in\#\ \textit{dom-m}\ N \wedge K \in \textit{set}\ (N \propto i) \wedge K \neq L \wedge$
$\textit{correctly-marked-as-binary}\ N\ (i,\ K,\ b)) \wedge$
$\textit{fst}\ `\#\ \textit{mset}\ (W\ L) = \textit{clause-to-update}\ L\ (M,\ N,\ D,\ NE,\ UE,\ \{\#\},\ \{\#\}))$›

lemma *correct-watching-init-correct-watching*:
‹$\textit{correct-watching-init}\ T \Longrightarrow \textit{correct-watching}\ T$›
⟨*proof*⟩

lemma *image-mset-Suc*: ‹$\textit{Suc}\ `\#\ \{\#C \in\#\ M.\ P\ C\#\} = \{\#C \in\#\ \textit{Suc}\ `\#\ M.\ P\ (C-1)\#\}$›
⟨*proof*⟩

lemma *correct-watching-init-add-unit*:
assumes ‹$\textit{correct-watching-init}\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W)$›
shows ‹$\textit{correct-watching-init}\ (M,\ N,\ D,\ \textit{add-mset}\ C\ NE,\ UE,\ Q,\ W)$›
⟨*proof*⟩

lemma *correct-watching-init-propagate*:
‹$\textit{correct-watching-init}\ ((L\ \#\ M,\ N,\ D,\ NE,\ UE,\ Q,\ W)) \longleftrightarrow$
$\textit{correct-watching-init}\ ((M,\ N,\ D,\ NE,\ UE,\ Q,\ W))$›
‹$\textit{correct-watching-init}\ ((M,\ N,\ D,\ NE,\ UE,\ \textit{add-mset}\ C\ Q,\ W)) \longleftrightarrow$
$\textit{correct-watching-init}\ ((M,\ N,\ D,\ NE,\ UE,\ Q,\ W))$›
⟨*proof*⟩

lemma *all-blits-are-in-problem-cons*[*simp*]:
‹$\textit{all-blits-are-in-problem-init}\ (\textit{Propagated}\ L\ i\ \#\ a,\ aa,\ ab,\ ac,\ ad,\ ae,\ b) \longleftrightarrow$
$\textit{all-blits-are-in-problem-init}\ (a,\ aa,\ ab,\ ac,\ ad,\ ae,\ b)$›
‹$\textit{all-blits-are-in-problem-init}\ (\textit{Decided}\ L\ \#\ a,\ aa,\ ab,\ ac,\ ad,\ ae,\ b) \longleftrightarrow$
$\textit{all-blits-are-in-problem-init}\ (a,\ aa,\ ab,\ ac,\ ad,\ ae,\ b)$›
‹$\textit{all-blits-are-in-problem-init}\ (a,\ aa,\ ab,\ ac,\ ad,\ \textit{add-mset}\ L\ ae,\ b) \longleftrightarrow$
$\textit{all-blits-are-in-problem-init}\ (a,\ aa,\ ab,\ ac,\ ad,\ ae,\ b)$›
‹$\textit{NO-MATCH None}\ y \Longrightarrow \textit{all-blits-are-in-problem-init}\ (a,\ aa,\ y,\ ac,\ ad,\ ae,\ b) \longleftrightarrow$
$\textit{all-blits-are-in-problem-init}\ (a,\ aa,\ \textit{None},\ ac,\ ad,\ ae,\ b)$›
‹$\textit{NO-MATCH}\ \{\#\}\ ae \Longrightarrow \textit{all-blits-are-in-problem-init}\ (a,\ aa,\ y,\ ac,\ ad,\ ae,\ b) \longleftrightarrow$
$\textit{all-blits-are-in-problem-init}\ (a,\ aa,\ y,\ ac,\ ad,\ \{\#\},\ b)$›
⟨*proof*⟩

lemma *correct-watching-init-cons*[*simp*]:
‹$\textit{NO-MATCH None}\ y \Longrightarrow \textit{correct-watching-init}\ ((a,\ aa,\ y,\ ac,\ ad,\ ae,\ b)) \longleftrightarrow$
$\textit{correct-watching-init}\ ((a,\ aa,\ \textit{None},\ ac,\ ad,\ ae,\ b))$›
‹$\textit{NO-MATCH}\ \{\#\}\ ae \Longrightarrow \textit{correct-watching-init}\ ((a,\ aa,\ y,\ ac,\ ad,\ ae,\ b)) \longleftrightarrow$
$\textit{correct-watching-init}\ ((a,\ aa,\ y,\ ac,\ ad,\ \{\#\},\ b))$›
⟨*proof*⟩

lemma *clause-to-update-mapsto-upd-notin*:
  assumes
    i: ‹$i \notin\# \mathit{dom\text{-}m}\ N$›
  shows
  ‹$\mathit{clause\text{-}to\text{-}update}\ L\ (M,\ N(i \hookrightarrow C'),\ C,\ NE,\ UE,\ WS,\ Q) =$
    $(\mathit{if}\ L \in \mathit{set}\ (\mathit{watched\text{-}l}\ C')$
     $\mathit{then}\ \mathit{add\text{-}mset}\ i\ (\mathit{clause\text{-}to\text{-}update}\ L\ (M,\ N,\ C,\ NE,\ UE,\ WS,\ Q))$
     $\mathit{else}\ (\mathit{clause\text{-}to\text{-}update}\ L\ (M,\ N,\ C,\ NE,\ UE,\ WS,\ Q)))$›
  ‹$\mathit{clause\text{-}to\text{-}update}\ L\ (M,\ \mathit{fmupd}\ i\ (C',\ b)\ N,\ C,\ NE,\ UE,\ WS,\ Q) =$
    $(\mathit{if}\ L \in \mathit{set}\ (\mathit{watched\text{-}l}\ C')$
     $\mathit{then}\ \mathit{add\text{-}mset}\ i\ (\mathit{clause\text{-}to\text{-}update}\ L\ (M,\ N,\ C,\ NE,\ UE,\ WS,\ Q))$
     $\mathit{else}\ (\mathit{clause\text{-}to\text{-}update}\ L\ (M,\ N,\ C,\ NE,\ UE,\ WS,\ Q)))$›
  ⟨*proof*⟩

lemma *correct-watching-init-add-clause*:
  assumes
    corr: ‹$\mathit{correct\text{-}watching\text{-}init}\ ((a,\ aa,\ \mathit{None},\ ac,\ ad,\ Q,\ b))$› **and**
    leC: ‹$2 \le \mathit{length}\ C$› **and**
    [*simp*]: ‹$i \notin\# \mathit{dom\text{-}m}\ aa$› **and**
    dist[*iff*]: ‹$C\ !\ 0 \neq C\ !\ \mathit{Suc}\ 0$›
  shows ‹$\mathit{correct\text{-}watching\text{-}init}$
        $((a,\ \mathit{fmupd}\ i\ (C,\ \mathit{red})\ aa,\ \mathit{None},\ ac,\ ad,\ Q,\ b$
          $(C\ !\ 0 := b\ (C\ !\ 0)\ @\ [(i,\ C\ !\ \mathit{Suc}\ 0,\ \mathit{length}\ C = 2)],$
           $C\ !\ \mathit{Suc}\ 0 := b\ (C\ !\ \mathit{Suc}\ 0)\ @\ [(i,\ C\ !\ 0,\ \mathit{length}\ C = 2)])))$›
⟨*proof*⟩

definition *rewatch*
  :: ‹$'v\ \mathit{clauses\text{-}l} \Rightarrow ('v\ \mathit{literal} \Rightarrow 'v\ \mathit{watched}) \Rightarrow ('v\ \mathit{literal} \Rightarrow 'v\ \mathit{watched})\ \mathit{nres}$›
where
‹$\mathit{rewatch}\ N\ W = \mathit{do}\ \{$
  $xs \leftarrow \mathit{SPEC}(\lambda xs.\ \mathit{set\text{-}mset}\ (\mathit{dom\text{-}m}\ N) \subseteq \mathit{set}\ xs \wedge \mathit{distinct}\ xs);$
  $\mathit{nfoldli}$
    xs
    $(\lambda\text{-}.\ \mathit{True})$
    $(\lambda i\ W.\ \mathit{do}\ \{$
      $\mathit{if}\ i \in\# \mathit{dom\text{-}m}\ N$
      $\mathit{then}\ \mathit{do}\ \{$
        $\mathit{ASSERT}(i \in\# \mathit{dom\text{-}m}\ N);$
        $\mathit{ASSERT}(\mathit{length}\ (N \propto i) \ge 2);$
        $\mathit{let}\ L1 = N \propto i\ !\ 0;$
        $\mathit{let}\ L2 = N \propto i\ !\ 1;$
        $\mathit{let}\ b = (\mathit{length}\ (N \propto i) = 2);$
        $\mathit{let}\ W = W(L1 := W\ L1\ @\ [(i,\ L2,\ b)]);$
        $\mathit{let}\ W = W(L2 := W\ L2\ @\ [(i,\ L1,\ b)]);$
        $\mathit{RETURN}\ W$
      $\}$
      $\mathit{else}\ \mathit{RETURN}\ W$
    $\})$
    W
  $\}$›

lemma *rewatch-correctness*:
  assumes [*simp*]: ‹$W = (\lambda\text{-}.\ [])$› **and**
    H[*dest*]: ‹$\bigwedge x.\ x \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{distinct}\ (N \propto x) \wedge \mathit{length}\ (N \propto x) \ge 2$›
  shows

$\langle$*rewatch N W* $\leq$ *SPEC*$(\lambda W.$ *correct-watching-init* $(M, N, C, NE, UE, Q, W))\rangle$
$\langle proof \rangle$

definition *state-wl-l-init-full* :: $\langle('v$ *twl-st-wl-init-full* $\times$ $'v$ *twl-st-l-init*$)$ *set*$\rangle$ **where**
$\langle$*state-wl-l-init-full* $= \{(S, S').\ (fst\ S, fst\ S') \in$ *state-wl-l None* $\wedge$
snd $S =$ *snd* $S'\}\rangle$

definition *added-only-watched* :: $\langle('v$ *twl-st-wl-init-full* $\times$ $'v$ *twl-st-wl-init*$)$ *set*$\rangle$ **where**
$\langle$*added-only-watched* $= \{(((M, N, D, NE, UE, Q, W), OC), ((M', N', D', NE', UE', Q'), OC')).$
$(M, N, D, NE, UE, Q) = (M', N', D', NE', UE', Q') \wedge OC = OC'\}\rangle$

definition *init-dt-wl-spec-full*
:: $\langle 'v$ *clause-l list* $\Rightarrow$ $'v$ *twl-st-wl-init* $\Rightarrow$ $'v$ *twl-st-wl-init-full* $\Rightarrow$ *bool*$\rangle$
where
$\langle$*init-dt-wl-spec-full* $C\ S\ T'' \longleftrightarrow$
$(\exists S'\ T\ T'.\ (S, S') \in$ *state-wl-l-init* $\wedge (T :: 'v$ *twl-st-wl-init*$, T') \in$ *state-wl-l-init* $\wedge$
init-dt-spec $C\ S'\ T' \wedge$ *correct-watching-init* $(fst\ T'') \wedge (T'', T) \in$ *added-only-watched*$)\rangle$

definition *init-dt-wl-full* :: $\langle 'v$ *clause-l list* $\Rightarrow$ $'v$ *twl-st-wl-init* $\Rightarrow$ $'v$ *twl-st-wl-init-full nres*$\rangle$ **where**
$\langle$*init-dt-wl-full CS S* $=$ *do*$\{$
$((M, N, D, NE, UE, Q), OC) \leftarrow$ *init-dt-wl CS S*;
$W \leftarrow$ *rewatch* N $(\lambda$-. $[])$;
RETURN $((M, N, D, NE, UE, Q, W), OC)$
$\}\rangle$

lemma *init-dt-wl-spec-rewatch-pre*:
assumes $\langle$*init-dt-wl-spec CS S T*$\rangle$ **and** $\langle N =$ *get-clauses-init-wl* $T\rangle$ **and** $\langle C \in\#$ *dom-m* $N\rangle$
shows $\langle$*distinct* $(N \propto C) \wedge$ *length* $(N \propto C) \geq 2\rangle$
$\langle proof \rangle$

lemma *init-dt-wl-full-init-dt-wl-spec-full*:
assumes $\langle$*init-dt-wl-pre CS S*$\rangle$
shows $\langle$*init-dt-wl-full CS S* $\leq$ *SPEC* (*init-dt-wl-spec-full CS S*)$\rangle$
$\langle proof \rangle$

end
theory *CDCL-Conflict-Minimisation*
imports
Watched-Literals-Watch-List-Domain
WB-More-Refinement
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* ⊨*as CNot* (*remove1-mset L C*)› **and** ‹*L* ∈# *C*› **and** ‹*N* + *U* ⊨*pm C*› **and**
‹*K* ∈# *remove1-mset L C* ⟹ *index-in-trail M K* < *index-in-trail M L*›
⟨*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 ⊨as CNot C›
shows ‹M ⊨as CNot E›
⟨*proof*⟩

lemma *rtranclp-minimize-conflict-support-entailed-trail*:
assumes ‹$(minimize\text{-}conflict\text{-}support\ M)^{**}$ C E› **and** ‹M ⊨as CNot C›
shows ‹M ⊨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 ⊨p add-mset L C› **and**
‹NU ⊨p add-mset (−L) D› **and**
‹⋀K'. K' ∈# C ⟹ NU ⊨p add-mset (−K') D›
shows ‹NU ⊨p D›
⟨*proof*⟩

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**
‹$\mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D = \mathit{filter\text{-}mset}\ (\lambda K.\ \mathit{index\text{-}in\text{-}trail}\ M\ K < \mathit{index\text{-}in\text{-}trail}\ M\ L)\ D$›

lemma *filter-to-poslev-uminus*[*simp*]:
‹$\mathit{filter\text{-}to\text{-}poslev}\ M\ (-L)\ D = \mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D$›
⟨*proof*⟩

lemma *filter-to-poslev-empty*[*simp*]:
‹$\mathit{filter\text{-}to\text{-}poslev}\ M\ L\ \{\#\} = \{\#\}$›
⟨*proof*⟩

lemma *filter-to-poslev-mono*:
‹$\mathit{index\text{-}in\text{-}trail}\ M\ K' \le \mathit{index\text{-}in\text{-}trail}\ M\ L \Longrightarrow$
$\mathit{filter\text{-}to\text{-}poslev}\ M\ K'\ D \subseteq\#\ \mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D$›
⟨*proof*⟩

lemma *filter-to-poslev-mono-entailement*:
‹$\mathit{index\text{-}in\text{-}trail}\ M\ K' \le \mathit{index\text{-}in\text{-}trail}\ M\ L \Longrightarrow$
$NU \models p\ \mathit{filter\text{-}to\text{-}poslev}\ M\ K'\ D \Longrightarrow NU \models p\ \mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D$›
⟨*proof*⟩

lemma *filter-to-poslev-mono-entailement-add-mset*:
‹$\mathit{index\text{-}in\text{-}trail}\ M\ K' \le \mathit{index\text{-}in\text{-}trail}\ M\ L \Longrightarrow$
$NU \models p\ \mathit{add\text{-}mset}\ J\ (\mathit{filter\text{-}to\text{-}poslev}\ M\ K'\ D) \Longrightarrow NU \models p\ \mathit{add\text{-}mset}\ J\ (\mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D)$›
⟨*proof*⟩

lemma *conflict-minimize-intermediate-step*:
assumes
‹$NU \models p\ \mathit{add\text{-}mset}\ L\ C$› **and**
K′-C: ‹$\bigwedge K'.\ K' \in\#\ C \Longrightarrow NU \models p\ \mathit{add\text{-}mset}\ (-K')\ D \lor K' \in\#\ D$›
shows ‹$NU \models p\ \mathit{add\text{-}mset}\ L\ D$›
⟨*proof*⟩

lemma *conflict-minimize-intermediate-step-filter-to-poslev*:
assumes
lev-K-L: ‹$\bigwedge K'.\ K' \in\#\ C \Longrightarrow \mathit{index\text{-}in\text{-}trail}\ M\ K' < \mathit{index\text{-}in\text{-}trail}\ M\ L$› **and**
NU-LC: ‹$NU \models p\ \mathit{add\text{-}mset}\ L\ C$› **and**
K′-C: ‹$\bigwedge K'.\ K' \in\#\ C \Longrightarrow NU \models p\ \mathit{add\text{-}mset}\ (-K')\ (\mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D) \lor$
$K' \in\#\ \mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D$›
shows ‹$NU \models p\ \mathit{add\text{-}mset}\ L\ (\mathit{filter\text{-}to\text{-}poslev}\ M\ L\ D)$›
⟨*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* = ‹$('v\ \mathit{literal} \times 'v\ \mathit{clause})\ \mathit{list}$›

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* =
$SPEC(\lambda(L, ana).\ L \in\#\ snd\ (hd\ analyse) \wedge tl\ ana = tl\ analyse \wedge ana \neq [] \wedge$
$hd\ ana = (fst\ (hd\ analyse),\ snd\ (hd\ (analyse)) - \{\#L\#\}))$›

definition *mark-failed-lits*
:: ‹- ⇒ *'v conflict-min-analyse* ⇒ *'v conflict-min-cach* ⇒ *'v conflict-min-cach nres*›
where
‹*mark-failed-lits NU analyse cach* = $SPEC(\lambda cach'.$
$(\forall L.\ cach'\ L = SEEN\text{-}REMOVABLE \longrightarrow cach\ L = SEEN\text{-}REMOVABLE))$›

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* ⟷
$(\forall L.\ -L \in lits\text{-}of\text{-}l\ M \longrightarrow cach\ (atm\text{-}of\ L) = SEEN\text{-}REMOVABLE \longrightarrow$
$set\text{-}mset\ NU \models p\ add\text{-}mset\ (-L)\ (filter\text{-}to\text{-}poslev\ M\ L\ D))$›

lemma *conflict-min-analysis-inv-update-removable*:
‹$no\text{-}dup\ M \Longrightarrow -L \in lits\text{-}of\text{-}l\ M \Longrightarrow$
$conflict\text{-}min\text{-}analysis\text{-}inv\ M\ (cach(atm\text{-}of\ L := SEEN\text{-}REMOVABLE))\ NU\ D \longleftrightarrow$
$conflict\text{-}min\text{-}analysis\text{-}inv\ M\ cach\ NU\ D \wedge set\text{-}mset\ NU \models p\ add\text{-}mset\ (-L)\ (filter\text{-}to\text{-}poslev\ M\ L\ D)$›
⟨*proof*⟩

lemma *conflict-min-analysis-inv-update-failed*:
‹$conflict\text{-}min\text{-}analysis\text{-}inv\ M\ cach\ NU\ D \Longrightarrow$
$conflict\text{-}min\text{-}analysis\text{-}inv\ M\ (cach(L := SEEN\text{-}FAILED))\ NU\ D$›
⟨*proof*⟩

fun *conflict-min-analysis-stack*
:: ‹(*'v*, *'a*) *ann-lits* ⇒ *'v clauses* ⇒ *'v clause* ⇒ *'v conflict-min-analyse* ⇒ *bool*›
where
‹$conflict\text{-}min\text{-}analysis\text{-}stack\ M\ NU\ D\ [] \longleftrightarrow True$› |
‹$conflict\text{-}min\text{-}analysis\text{-}stack\ M\ NU\ D\ ((L, E)\ \#\ []) \longleftrightarrow True$› |
‹$conflict\text{-}min\text{-}analysis\text{-}stack\ M\ NU\ D\ ((L, E)\ \#\ (L', E')\ \#\ analyse) \longleftrightarrow$
$(\exists C.\ set\text{-}mset\ NU \models p\ add\text{-}mset\ (-L')\ C \wedge$
$(\forall K\in\#C-add\text{-}mset\ L\ E'.\ set\text{-}mset\ NU \models p\ (filter\text{-}to\text{-}poslev\ M\ L'\ D) + \{\#-K\#\} \vee$
$K \in\#\ filter\text{-}to\text{-}poslev\ M\ L'\ D) \wedge$
$(\forall K\in\#C.\ index\text{-}in\text{-}trail\ M\ K < index\text{-}in\text{-}trail\ M\ L') \wedge$
$E' \subseteq\#\ C) \wedge$
$-L' \in lits\text{-}of\text{-}l\ M \wedge$
$index\text{-}in\text{-}trail\ M\ L < index\text{-}in\text{-}trail\ M\ L' \wedge$
$conflict\text{-}min\text{-}analysis\text{-}stack\ M\ NU\ D\ ((L', E')\ \#\ analyse)$›

lemma *conflict-min-analysis-stack-change-hd*:
‹$conflict\text{-}min\text{-}analysis\text{-}stack\ M\ NU\ D\ ((L, E)\ \#\ ana) \Longrightarrow$
$conflict\text{-}min\text{-}analysis\text{-}stack\ M\ NU\ D\ ((L, E')\ \#\ ana)$›
⟨*proof*⟩

fun *conflict-min-analysis-stack-hd*
:: ‹(*'v*, *'a*) *ann-lits* ⇒ *'v clauses* ⇒ *'v clause* ⇒ *'v conflict-min-analyse* ⇒ *bool*›

where
‹*conflict-min-analysis-stack-hd M NU D* [] ⟷ *True*› |
‹*conflict-min-analysis-stack-hd M NU D* ((*L*, *E*) # -) ⟷
  (∃ *C*. *set-mset NU* ⊨*p add-mset* (−*L*) *C* ∧
  (∀ *K*∈#*C*. *index-in-trail M K* < *index-in-trail M L*) ∧ *E* ⊆# *C* ∧ −*L* ∈ *lits-of-l M* ∧
  (∀ *K*∈#*C*−*E*. *set-mset NU* ⊨*p* (*filter-to-poslev M L D*) + {#−*K*#} ∨ *K* ∈# *filter-to-poslev M L D*))›

lemma *conflict-min-analysis-stack-tl*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹ *conflict-min-analysis-stack M NU D* (*tl analyse*)›
⟨*proof*⟩

definition *lit-redundant-inv*
:: ‹(′*v*, ′*v clause*) *ann-lits* ⇒ ′*v clauses* ⇒ ′*v clause* ⇒ ′*v conflict-min-analyse* ⇒
  ′*v conflict-min-cach* × ′*v conflict-min-analyse* × *bool* ⇒ *bool*› **where**
‹*lit-redundant-inv M NU D init-analyse* = (λ(*cach*, *analyse*, *b*).
    conflict-min-analysis-inv M cach NU D ∧
    (*analyse* ≠ [] ⟶ *fst* (*hd init-analyse*) = *fst* (*last analyse*)) ∧
    (*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analyse*))) = *SEEN-REMOVABLE*) ∧
    conflict-min-analysis-stack M NU D analyse ∧
    conflict-min-analysis-stack-hd M NU D analyse)›

definition *lit-redundant-rec* :: ‹(′*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* =
  $WHILE_T$
   (λ(*cach*, *analyse*, *b*). *analyse* ≠ [])
   (λ(*cach*, *analyse*, *b*). *do* {
     ASSERT(*analyse* ≠ []);
     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 {
         C ← *get-propagation-reason M* (−*L*);
         case C of
           Some 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* $\longrightarrow$ *NU* $\models$*pm add-mset* ($-L$) (*filter-to-poslev M L D*)) $\wedge$
conflict-min-analysis-inv M cach NU D)›

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*) $\in$ *set M*› **and**
‹*conflict-min-analysis-inv M cach* (*N* + *NE* + *U* + *UE*) *D*› **and**
L-D: ‹*L* $\in\#$ *D*› **and**
M-D: ‹*M* $\models$*as CNot D*›
shows
‹*lit-redundant-rec M* (*N* + *U*) *D cach init-analysis* $\leq$
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* $\longrightarrow$ *NU* $\models$*pm add-mset* ($-L$) (*filter-to-poslev M L D*)) $\wedge$
conflict-min-analysis-inv M cach NU D)›

definition *literal-redundant* **where**
‹*literal-redundant M NU D cach L* = *do* {
ASSERT($-L \in$ *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 $\leftarrow$ *get-propagation-reason M* ($-L$);
case C of
Some C $\Rightarrow$ *lit-redundant-rec M NU D cach* [(*L*, *C* $-$ {#$-L$#})]
| *None* $\Rightarrow$ *do* {
RETURN (*cach*, [], *False*)
}
}
}›

lemma *true-clss-cls-add-self*: ‹*NU* $\models$*p D'* + *D'* $\longleftrightarrow$ *NU* $\models$*p D'*›
⟨*proof*⟩

lemma *true-clss-cls-add-add-mset-self*: ‹*NU* $\models$*p add-mset L* (*D'* + *D'*) $\longleftrightarrow$ *NU* $\models$*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* $\leq$ *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 ∈# 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 ∈# 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 ∈# 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* {
$S \leftarrow WHILE^{\lambda(i,\ xs).\ i \le length\ xs \wedge (\forall x \in set\ (take\ i\ xs).\ x = e) \wedge length\ xs = length\ ys}$
(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*) *list* ⇒ ′*v literal* × (*nat* × *nat*) *list*› **where**
‹*get-literal-and-remove-of-analyse-wl* C *analyse* =
(*let* (i, j) = *last analyse in*

$(C \,!\, j,\ analyse[length\ analyse - 1 := (i, j + 1)]))$›

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**
‹$lit\text{-}redundant\text{-}rec\text{-}wl\text{-}ref\ NU\ analyse \longleftrightarrow$
$(\forall (i, j) \in set\ analyse.\ j \le length\ (NU \propto i) \wedge i \in\# dom\text{-}m\ NU \wedge j \ge 1 \wedge i > 0)$›

definition *lit-redundant-rec-wl-inv* **where**
‹$lit\text{-}redundant\text{-}rec\text{-}wl\text{-}inv\ M\ NU\ D = (\lambda(cach, analyse, b).\ lit\text{-}redundant\text{-}rec\text{-}wl\text{-}ref\ NU\ analyse)$›

context *isasat-input-ops*
begin

definition (**in** $-$) *lit-redundant-rec-wl* :: ‹$('v, nat)\ ann\text{-}lits \Rightarrow 'v\ clauses\text{-}l \Rightarrow 'v\ clause \Rightarrow$
$\text{-} \Rightarrow \text{-} \Rightarrow \text{-} \Rightarrow$
$(\text{-} \times \text{-} \times bool)\ nres$›
where
‹$lit\text{-}redundant\text{-}rec\text{-}wl\ M\ NU\ D\ cach\ analysis\ \text{-} =$
$WHILE_T{}^{lit\text{-}redundant\text{-}rec\text{-}wl\text{-}inv\ M\ NU\ D}$
$(\lambda(cach, analyse, b).\ analyse \ne [])$
$(\lambda(cach, analyse, b).\ do\ \{$
$ASSERT(analyse \ne []);$
$ASSERT(fst\ (last\ analyse) \in\# dom\text{-}m\ NU);$
$let\ C = NU \propto fst\ (last\ analyse);$
$ASSERT(length\ C \ge 1);$
$let\ i = snd\ (last\ analyse);$
$ASSERT(C!0 \in lits\text{-}of\text{-}l\ M);$
$if\ i \ge length\ C$
$then$
$RETURN(cach\ (atm\text{-}of\ (C \,!\, 0) := SEEN\text{-}REMOVABLE), butlast\ analyse, True)$
$else\ do\ \{$
$let\ (L, analyse) = get\text{-}literal\text{-}and\text{-}remove\text{-}of\text{-}analyse\text{-}wl\ C\ analyse;$
$ASSERT(-L \in lits\text{-}of\text{-}l\ M);$
$b \leftarrow RES\ (UNIV);$
$if\ (get\text{-}level\ M\ L = 0 \vee cach\ (atm\text{-}of\ L) = SEEN\text{-}REMOVABLE \vee L \in\# D)$
$then\ RETURN\ (cach, analyse, False)$
$else\ if\ b \vee cach\ (atm\text{-}of\ L) = SEEN\text{-}FAILED$
$then\ do\ \{$
$cach \leftarrow mark\text{-}failed\text{-}lits\text{-}wl\ NU\ analyse\ cach;$
$RETURN\ (cach, [], False)$
$\}$
$else\ do\ \{$
$C \leftarrow get\text{-}propagation\text{-}reason\ M\ (-L);$
$case\ C\ of$
$Some\ C \Rightarrow RETURN\ (cach, analyse\ @\ [(C, 1)], False)$
$|\ None \Rightarrow do\ \{$
$cach \leftarrow mark\text{-}failed\text{-}lits\text{-}wl\ NU\ analyse\ cach;$
$RETURN\ (cach, [], False)$
$\}$
$\}$

}
})
(*cach*, *analysis*, *False*)›

fun *convert-analysis-l* **where**
‹*convert-analysis-l NU* (*i*, *j*) = (−*NU* ∝ *i* ! *0*, *mset* (*drop j* (*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 *lit-redundant-rec-wl*:
fixes *S* :: ‹*nat twl-st-wl*› **and** *S′* :: ‹*nat twl-st-l*› **and** *S″* :: ‹*nat twl-st*› **and** *NU M analyse*
defines
[*simp*]: ‹*S‴* ≡ *state*$_W$*-of S″*›
defines
‹*M* ≡ *get-trail-wl S*› **and**
M′: ‹*M′* ≡ *trail S‴*› **and**
NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
NU′: ‹*NU′* ≡ *mset* ‘# *ran-mf NU*› **and**
‹*analyse′* ≡ *convert-analysis-list NU analyse*›
assumes
S-S′: ‹(*S*, *S′*) ∈ *state-wl-l None*› **and**
S′-S″: ‹(*S′*, *S″*) ∈ *twl-st-l None*› **and**
bounds-init: ‹*lit-redundant-rec-wl-ref NU analyse*› **and**
struct-invs: ‹*twl-struct-invs S″*› **and**
add-inv: ‹*twl-list-invs S′*›
shows
‹*lit-redundant-rec-wl M NU D cach analyse lbv* ≤ ⇓
(*Id* $\times_r$ {(*analyse*, *analyse′*). *analyse′* = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse} $\times_r$ *bool-rel*)
(*lit-redundant-rec M′ NU′ D cach analyse′*)›
(**is** ‹- ≤ ⇓ (- $\times_r$ *?A* $\times_r$ -) -› **is** ‹- ≤ ⇓ *?R* -›)
⟨*proof*⟩

definition *literal-redundant-wl* **where**
‹*literal-redundant-wl M NU D cach L lbd* = *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 ⇒ *lit-redundant-rec-wl M NU D cach* [(*C*, *1*)] *lbd*
| *None* ⇒ *do* {
RETURN (*cach*, [], *False*)
}
}
}›

lemma *literal-redundant-wl-literal-redundant*:
fixes *S* :: ‹*nat twl-st-wl*› **and** *S'* :: ‹*nat twl-st-l*› **and** *S''* :: ‹*nat twl-st*› **and** *NU M*
defines
[*simp*]: ‹*S'''* ≡ $state_W$*-of S''*›
defines
‹*M* ≡ *get-trail-wl S*› **and**
M': ‹*M'* ≡ *trail S'''*› **and**
NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
NU': ‹*NU'* ≡ *mset ‘# ran-mf NU*›
assumes
S-S': ‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
S'-S'': ‹(*S'*, *S''*) ∈ *twl-st-l None*› **and**
‹*M* ≡ *get-trail-wl S*› **and**
M': ‹*M'* ≡ *trail S'''*› **and**
NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
NU': ‹*NU'* ≡ *mset ‘# ran-mf NU*›
assumes
struct-invs: ‹*twl-struct-invs S''*› **and**
add-inv: ‹*twl-list-invs S'*› **and**
L-D: ‹*L* ∈# *D*› **and**
M-D: ‹*M* ⊨*as CNot D*›
shows
‹*literal-redundant-wl M NU D cach L lbd* ≤ ⇓
(*Id* $\times_r$ {(*analyse*, *analyse'*). *analyse'* = *convert-analysis-list NU analyse* ∧
(∀ (*i*, *j*)∈ *set analyse*. *j* ≤ *length* (*NU*∝*i*) ∧ *i* ∈# *dom-m NU* ∧ *j* ≥ *1* ∧ *i* > *0*)} $\times_r$ *bool-rel*)
(*literal-redundant M' NU' D cach L*)›
(**is** ‹- ≤ ⇓ (- $\times_r$ *?A* $\times_r$ -) -› **is** ‹- ≤ ⇓ *?R* -›)
⟨*proof*⟩

definition *mark-failed-lits-stack-inv* **where**
‹*mark-failed-lits-stack-inv NU analyse* = (λ*cach*.
(∀ (*i*, *j*) ∈ *set analyse*. *j* ≤ *length* (*NU* ∝ *i*) ∧ *i* ∈# *dom-m NU* ∧ *j* ≥ *1* ∧ *i* > *0*))›

We mark all the literals from the current literal stack as failed, since every minimisation call will find the same minimisation problem.

definition (**in** *isasat-input-ops*) *mark-failed-lits-stack* **where**
‹*mark-failed-lits-stack NU analyse cach* = *do* {
(-, *cach*) ← $WHILE_T$$^{λ(\text{-},\ cach).\ mark\text{-}failed\text{-}lits\text{-}stack\text{-}inv\ NU\ analyse\ cach}$
(λ(*i*, *cach*). *i* < *length analyse*)
(λ(*i*, *cach*). *do* {
ASSERT(*i* < *length analyse*);
let (*cls-idx*, *idx*) = *analyse* ! *i*;
ASSERT(*atm-of* (*NU* ∝ *cls-idx* ! (*idx* − *1*)) ∈# $\mathcal{A}_{in}$);
RETURN (*i*+*1*, *cach* (*atm-of* (*NU* ∝ *cls-idx* ! (*idx* − *1*)) := *SEEN-FAILED*))
})
(*0*, *cach*);
RETURN cach
}›

lemma *mark-failed-lits-stack-mark-failed-lits-wl*:
shows
‹(*uncurry2 mark-failed-lits-stack*, *uncurry2 mark-failed-lits-wl*) ∈
[λ((*NU*, *analyse*), *cach*). *literals-are-in-*$\mathcal{L}_{in}$*-mm* (*mset ‘# ran-mf NU*) ∧
mark-failed-lits-stack-inv NU analyse cach$]_f$

$Id \times_f Id \times_f Id \rightarrow \langle Id \rangle nres\text{-}rel \rangle$
$\langle proof \rangle$

end

end