Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

May 31, 2016

Contents

Chapter 1

More Standard Theorems

This chapter contains additional lemmas built on top of HOL.

end

1.1 Transitions

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

theory *Wellfounded-More*
imports *Main*

begin

1.1.1 More theorems about Closures

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

lemma *tranclp-mono-explicit*:
$r^{++}\ a\ b \Longrightarrow r \leq s \Longrightarrow s^{++}\ a\ b$
⟨*proof*⟩

lemma *tranclp-mono*:
assumes *mono*: $r \leq s$
shows $r^{++} \leq s^{++}$
⟨*proof*⟩

lemma *tranclp-idemp-rel*:
$R^{++++}\ a\ b \longleftrightarrow R^{++}\ a\ b$
⟨*proof*⟩

Equivalent of the theorem *rtranclp-idemp*

lemma *trancl-idemp*: $(r^+)^+ = r^+$
⟨*proof*⟩

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

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

lemma *rtranclp-unfold*: *rtranclp* $r\ a\ b \longleftrightarrow (a = b \vee$ *tranclp* $r\ a\ b)$

⟨*proof*⟩

lemma *tranclp-unfold-end*: *tranclp r a b* $\longleftrightarrow$ ($\exists$ *a'*. *rtranclp r a a'* $\wedge$ *r a' b*)
⟨*proof*⟩

Near duplicate of theorem *tranclpD*:

lemma *tranclp-unfold-begin*: *tranclp r a b* $\longleftrightarrow$ ($\exists$ *a'*. *r a a'* $\wedge$ *rtranclp r a' b*)
⟨*proof*⟩

lemma *trancl-set-tranclp*: $(a,\ b) \in \{(b,a).\ P\ a\ b\}^+ \longleftrightarrow P^{++}\ b\ a$
⟨*proof*⟩

lemma *tranclp-rtranclp-rtranclp-rel*: $R^{++**}\ a\ b \longleftrightarrow R^{**}\ a\ b$
⟨*proof*⟩

lemma *tranclp-rtranclp-rtranclp*[*simp*]: $R^{++**} = R^{**}$
⟨*proof*⟩

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

lemma *rtranclp-and-rtranclp-left*: $(\lambda\ a\ b.\ P\ a\ b \wedge Q\ a\ b)^{**}\ S\ T \Longrightarrow P^{**}\ S\ T$
⟨*proof*⟩

1.1.2 Full Transitions

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

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

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

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

We define output notations only for printing:

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

lemma *rtranclp-full1I*:
$R^{**}\ a\ b \Longrightarrow$ *full1 R b c* $\Longrightarrow$ *full1 R a c*
⟨*proof*⟩

lemma *tranclp-full1I*:
$R^{++}\ a\ b \Longrightarrow$ *full1 R b c* $\Longrightarrow$ *full1 R a c*
⟨*proof*⟩

lemma *rtranclp-fullI*:
$R^{**}\ a\ b \Longrightarrow$ *full R b c* $\Longrightarrow$ *full R a c*
⟨*proof*⟩

lemma *tranclp-full-full1I*:

R^{++} a $b \Longrightarrow$ *full* R b $c \Longrightarrow$ *full1* R a c
⟨*proof*⟩

lemma *full-fullI*:
R a $b \Longrightarrow$ *full* R b $c \Longrightarrow$ *full1* R a c
⟨*proof*⟩

lemma *full-unfold*:
full r S $S' \longleftrightarrow ((S = S' \wedge$ *no-step* r $S') \vee$ *full1* r S $S')$
⟨*proof*⟩

lemma *full1-is-full*[*intro*]: *full1* R S $T \Longrightarrow$ *full* R S T
⟨*proof*⟩

lemma *not-full1-rtranclp-relation*: $\neg$*full1* R^{**} a b
⟨*proof*⟩

lemma *not-full-rtranclp-relation*: $\neg$*full* R^{**} a b
⟨*proof*⟩

lemma *full1-tranclp-relation-full*:
full1 R^{++} a $b \longleftrightarrow$ *full1* R a b
⟨*proof*⟩

lemma *full-tranclp-relation-full*:
full R^{++} a $b \longleftrightarrow$ *full* R a b
⟨*proof*⟩

lemma *rtranclp-full1-eq-or-full1*:
(*full1* $R)^{**}$ a $b \longleftrightarrow (a = b \vee$ *full1* R a $b)$
⟨*proof*⟩

lemma *tranclp-full1-full1*:
(*full1* $R)^{++}$ a $b \longleftrightarrow$ *full1* R a b
⟨*proof*⟩

1.1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:
assumes *wf*:*wf* $\{(x, y).\ R\ y\ x\}$
shows $\exists\, b.\ R^{**}$ a $b \wedge$ *no-step* R b
⟨*proof*⟩

lemma *wf-exists-normal-form-full*:
assumes *wf*: *wf* $\{(x, y).\ R\ y\ x\}$
shows $\exists\, b.$ *full* R a b
⟨*proof*⟩

1.1.4 More Well-Foundedness

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

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

lemma *wf-if-measure-in-wf*:

$wf\ R \Longrightarrow (\bigwedge a\ b.\ (a,\ b) \in S \Longrightarrow (\nu\ a,\ \nu\ b){\in}R) \Longrightarrow wf\ S$
⟨*proof*⟩

lemma *wfP-if-measure*: **fixes** $f :: 'a \Rightarrow nat$
shows $(\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow f\ y < f\ x) \Longrightarrow wf\ \{(y,x).\ P\ x \wedge g\ x\ y\}$
⟨*proof*⟩

lemma *wf-if-measure-f*:
assumes $wf\ r$
shows $wf\ \{(b,\ a).\ (f\ b,\ f\ a) \in r\}$
⟨*proof*⟩

lemma *wf-wf-if-measure'*:
assumes $wf\ r$ **and** H: $\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y,\ f\ x) \in r$
shows $wf\ \{(y,x).\ P\ x \wedge g\ x\ y\}$
⟨*proof*⟩

lemma *wf-lex-less*: $wf\ (lex\ \{(a,\ b).\ (a{::}nat) < b\})$
⟨*proof*⟩

lemma *wfP-if-measure2*: **fixes** $f :: 'a \Rightarrow nat$
shows $(\bigwedge x\ y.\ P\ x\ y \Longrightarrow g\ x\ y \Longrightarrow f\ x < f\ y) \Longrightarrow wf\ \{(x,y).\ P\ x\ y \wedge g\ x\ y\}$
⟨*proof*⟩

lemma *lexord-on-finite-set-is-wf*:
assumes
P-finite: $\bigwedge U.\ P\ U \longrightarrow U \in A$ **and**
finite: *finite* A **and**
wf: $wf\ R$ **and**
trans: *trans* R
shows $wf\ \{(T,\ S).\ (P\ S \wedge P\ T) \wedge (T,\ S) \in lexord\ R\}$
⟨*proof*⟩

lemma *wf-fst-wf-pair*:
assumes $wf\ \{(M',\ M).\ R\ M'\ M\}$
shows $wf\ \{((M',\ N'),\ (M,\ N)).\ R\ M'\ M\}$
⟨*proof*⟩

lemma *wf-snd-wf-pair*:
assumes $wf\ \{(M',\ M).\ R\ M'\ M\}$
shows $wf\ \{((M',\ N'),\ (M,\ N)).\ R\ N'\ N\}$
⟨*proof*⟩

lemma *wf-if-measure-f-notation2*:
assumes $wf\ r$
shows $wf\ \{(b,\ h\ a)|b\ a.\ (f\ b,\ f\ (h\ a)) \in r\}$
⟨*proof*⟩

lemma *wf-wf-if-measure'-notation2*:
assumes $wf\ r$ **and** H: $\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y,\ f\ (h\ x)) \in r$
shows $wf\ \{(y,h\ x)|\ y\ x.\ P\ x \wedge g\ x\ y\}$
⟨*proof*⟩

end
theory *List-More*

imports *Main ../lib/Multiset-More*
begin

Sledgehammer parameters

sledgehammer-params[*debug*]

1.2 Various Lemmas

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

thm *nat-less-induct*
lemma *nat-less-induct-case*[*case-names 0 Suc*]:
  assumes
    $P\ 0$ **and**
    $\bigwedge n.\ (\forall m < Suc\ n.\ P\ m) \Longrightarrow P\ (Suc\ n)$
  shows $P\ n$
  ⟨*proof*⟩

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

lemma *if-0-1-ge-0*[*simp*]:
  $0 < (if\ P\ then\ a\ else\ (0::nat)) \longleftrightarrow P \wedge 0 < a$
  ⟨*proof*⟩

Bounded function have not yet been defined in Isabelle.

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

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

lemma *not-bounded-nat-exists-larger*:
  fixes $f :: nat \Rightarrow nat$
  assumes *unbound*: *unbounded f*
  shows $\exists n.\ f\ n > m \wedge n > n_0$
⟨*proof*⟩

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

lemma *bounded-const-product*:
  fixes $k :: nat$ **and** $f :: nat \Rightarrow nat$
  assumes $k > 0$
  shows $bounded\ f \longleftrightarrow bounded\ (\lambda i.\ k * f\ i)$
  ⟨*proof*⟩

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

lemma *bounded-finite-linorder*:
  fixes $f :: 'a \Rightarrow 'a :: \{finite, linorder\}$
  shows *bounded f*
⟨*proof*⟩

1.3 More List

1.3.1 *upt*

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

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

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

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

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

lemma *take-upt-bound-minus*[*simp*]:
assumes $i \leq n - m$
shows $take\ i\ [m..<n] = [m\ ..<m+i]$
⟨*proof*⟩

lemma *append-cons-eq-upt*:
assumes $A\ @\ B = [m..<n]$
shows $A = [m\ ..<m+length\ A]$ **and** $B = [m + length\ A..<n]$
⟨*proof*⟩

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

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

⟨*proof*⟩

A more restrictive version holds:

lemma $B \neq [] \Longrightarrow A\ @\ B = [m..<\ n] \longleftrightarrow A = [m\ ..<m+length\ A] \wedge B = [m + length\ A..<n]$
(**is** $?P \Longrightarrow ?A = ?B$)
⟨*proof*⟩

lemma *append-cons-eq-upt-length-i*:
assumes $A\ @\ i\ \#\ B = [m..<n]$
shows $A = [m\ ..<i]$
⟨*proof*⟩

lemma *append-cons-eq-upt-length*:
assumes $A\ @\ i\ \#\ B = [m..<n]$
shows $length\ A = i - m$
⟨*proof*⟩

lemma *append-cons-eq-upt-length-i-end*:
assumes $A\ @\ i\ \#\ B = [m..<n]$
shows $B = [Suc\ i\ ..<n]$
⟨*proof*⟩

lemma *Max-n-upt*: $Max\ (insert\ 0\ \{Suc\ 0..<n\}) = n - Suc\ 0$
⟨*proof*⟩

lemma *upt-decomp-lt*:
assumes H: $xs @ i \# ys @ j \# zs = [m ..< n]$
shows $i < j$
⟨*proof*⟩

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

lemma *length-list-Suc-0*:
$length\ W = Suc\ 0 \longleftrightarrow (\exists L.\ W = [L])$
⟨*proof*⟩

lemma *length-list-2*: $length\ S = 2 \longleftrightarrow (\exists a\ b.\ S = [a, b])$
⟨*proof*⟩

lemma *finite-bounded-list*:
fixes $b :: nat$
shows $finite\ \{xs.\ length\ xs < s \wedge (\forall i< length\ xs.\ xs\ !\ i < b)\}$ (**is** $finite\ (?S\ s)$)
⟨*proof*⟩

1.3.2 Lexicographic Ordering

lemma *lexn-Suc*:
$(x \# xs, y \# ys) \in lexn\ r\ (Suc\ n) \longleftrightarrow$
$(length\ xs = n \wedge length\ ys = n) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in lexn\ r\ n))$
⟨*proof*⟩

lemma *lexn-n*:
$n > 0 \Longrightarrow (x \# xs, y \# ys) \in lexn\ r\ n \longleftrightarrow$
$(length\ xs = n-1 \wedge length\ ys = n-1) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in lexn\ r\ (n - 1)))$
⟨*proof*⟩

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

lemma *lexn2-conv*:
$([a, b], [c, d]) \in lexn\ r\ 2 \longleftrightarrow (a, c) \in r \vee (a = c \wedge (b, d) \in r)$
⟨*proof*⟩

lemma *lexn3-conv*:
$([a, b, c], [a', b', c']) \in lexn\ r\ 3 \longleftrightarrow$
$(a, a') \in r \vee (a = a' \wedge (b, b') \in r) \vee (a = a' \wedge b = b' \wedge (c, c') \in r)$
⟨*proof*⟩

1.3.3 Remove

More lemmas about remove

lemma *remove1-Nil*:
$remove1\ (-\ L)\ W = [] \longleftrightarrow (W = [] \vee W = [-L])$
⟨*proof*⟩

lemma *remove1-mset-single-add*:
$a \neq b \Longrightarrow remove1\text{-}mset\ a\ (\{\#b\#\} + C) = \{\#b\#\} + remove1\text{-}mset\ a\ C$
$remove1\text{-}mset\ a\ (\{\#a\#\} + C) = C$
⟨*proof*⟩

Remove under condition

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

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

lemma *remove1 x xs* = *remove1-cond* ((*op* =) *x*) *xs*
  ⟨*proof*⟩

lemma *mset-map-mset-remove1-cond*:
  mset (*map mset* (*remove1-cond* (λL. *mset L* = *mset a*) *C*)) =
    remove1-mset (*mset a*) (*mset* (*map mset C*))
  ⟨*proof*⟩

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

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

lemma *removeAll x xs* = *removeAll-cond* ((*op* =) *x*) *xs*
  ⟨*proof*⟩

lemma *removeAll-cond P xs* = *filter* (λx. $\neg P\ x$) *xs*
  ⟨*proof*⟩

lemma *mset-map-mset-removeAll-cond*:
  mset (*map mset* (*removeAll-cond* (λb. *mset b* = *mset a*) *C*))
 = *removeAll-mset* (*mset a*) (*mset* (*map mset C*))
  ⟨*proof*⟩

Filter

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

1.3.4 Multisets

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

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

lemma *union-mset-list*:
  mset xs #$\cup$ *mset ys* = *mset* (*union-mset-list xs ys*)
⟨*proof*⟩

lemma *size-le-Suc-0-iff*: *size M* $\leq$ *Suc 0* $\longleftrightarrow$ (($\exists$ *a b*. *M* = {#*a*#}) $\vee$ *M* = {#})
  ⟨*proof*⟩

lemma *size-2-iff*: *size M* = *2* $\longleftrightarrow$ ($\exists$ *a b*. *M* = {#*a*, *b*#})
  ⟨*proof*⟩

lemma *remove1-mset-eqE*:
  remove1-mset L *x1* $= M \Longrightarrow$
    $(L \in\# \ x1 \Longrightarrow x1 = M + \{\#L\#\} \Longrightarrow P) \Longrightarrow$
    $(L \notin\# \ x1 \Longrightarrow x1 = M \Longrightarrow P) \Longrightarrow$
  P
  $\langle proof \rangle$

lemma *subset-eq-mset-single-iff*: $x2 \subseteq\# \{\#L\#\} \longleftrightarrow x2 = \{\#\} \vee x2 = \{\#L\#\}$
  $\langle proof \rangle$

end

Chapter 2

Definition of Entailment

This chapter defines various form of entailment.

end

2.1 Clausal Logic

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

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

2.1.1 Literals

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

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

abbreviation *is-neg* :: $'a$ *literal* $\Rightarrow$ *bool* **where** *is-neg* $L \equiv \neg$ *is-pos* L

lemma *Pos-atm-of-iff* [*simp*]: *Pos* (*atm-of* L) = $L \longleftrightarrow$ *is-pos* L
  ⟨*proof*⟩

lemma *Neg-atm-of-iff* [*simp*]: *Neg* (*atm-of* L) = $L \longleftrightarrow$ *is-neg* L
  ⟨*proof*⟩

lemma *ex-lit-cases*: $(\exists L.\ P\ L) \longleftrightarrow (\exists A.\ P\ (Pos\ A) \lor P\ (Neg\ A))$
  ⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

lemma
  uminus-Pos[*simp*]: $- \ Pos\ A = Neg\ A$ **and**
  uminus-Neg[*simp*]: $- \ Neg\ A = Pos\ A$
  ⟨*proof*⟩

lemma *atm-of-uminus*[*simp*]:
  $atm\text{-}of\ (-L) = atm\text{-}of\ L$
  ⟨*proof*⟩

lemma *uminus-of-uminus-id*[*simp*]:
  $- (- (x:: 'v\ literal)) = x$
  ⟨*proof*⟩

lemma *uminus-not-id*[*simp*]:
  $x \neq - (x:: 'v\ literal)$
  ⟨*proof*⟩

lemma *uminus-not-id'*[*simp*]:
  $- x \neq (x:: 'v\ literal)$
  ⟨*proof*⟩

lemma *uminus-eq-inj*[*iff*]:
  $-(a::'v\ literal) = -b \longleftrightarrow a = b$
  ⟨*proof*⟩

lemma *uminus-lit-swap*:
  $(a::'a\ literal) = -b \longleftrightarrow -a = b$
  ⟨*proof*⟩

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

definition *less-literal* :: *'a literal* ⇒ *'a literal* ⇒ *bool* **where**
  $less\text{-}literal\ L\ M \longleftrightarrow atm\text{-}of\ L < atm\text{-}of\ M \lor atm\text{-}of\ L \leq atm\text{-}of\ M \land is\text{-}neg\ L < is\text{-}neg\ M$

definition *less-eq-literal* :: *'a literal* ⇒ *'a literal* ⇒ *bool* **where**
  $less\text{-}eq\text{-}literal\ L\ M \longleftrightarrow atm\text{-}of\ L < atm\text{-}of\ M \lor atm\text{-}of\ L \leq atm\text{-}of\ M \land is\text{-}neg\ L \leq is\text{-}neg\ M$

instance
  ⟨*proof*⟩

end

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

instance
  ⟨*proof*⟩

end

lemma *pos-less-neg*[*simp*]: $Pos\ A < Neg\ A$
  ⟨*proof*⟩

lemma *pos-less-pos-iff* [*simp*]: *Pos* A < *Pos* $B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *pos-less-neg-iff* [*simp*]: *Pos* A < *Neg* $B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *neg-less-pos-iff* [*simp*]: *Neg* A < *Pos* $B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *neg-less-neg-iff* [*simp*]: *Neg* A < *Neg* $B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *pos-le-neg*[*simp*]: *Pos* $A \leq$ *Neg* A
⟨*proof*⟩

lemma *pos-le-pos-iff* [*simp*]: *Pos* $A \leq$ *Pos* $B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *pos-le-neg-iff* [*simp*]: *Pos* $A \leq$ *Neg* $B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *neg-le-pos-iff* [*simp*]: *Neg* $A \leq$ *Pos* $B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *neg-le-neg-iff* [*simp*]: *Neg* $A \leq$ *Neg* $B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *leq-imp-less-eq-atm-of*: $L \leq M \Longrightarrow$ *atm-of* $L \leq$ *atm-of* M
⟨*proof*⟩

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

instance
⟨*proof*⟩

end

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

instance
⟨*proof*⟩

end

2.1.2 Clauses

Clauses are (finite) multisets of literals.

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

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

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

$\langle proof \rangle$

lemma *Max-in-lits*: $C \neq \{\#\} \Longrightarrow Max\ (set\text{-}mset\ C) \in\#\ C$
$\langle proof \rangle$

lemma *Max-atm-of-set-mset-commute*: $C \neq \{\#\} \Longrightarrow Max\ (atm\text{-}of\ \text{`}\ set\text{-}mset\ C) = atm\text{-}of\ (Max\ (set\text{-}mset\ C))$
$\langle proof \rangle$

lemma *Max-pos-neg-less-multiset*:
assumes *max*: $Max\ (set\text{-}mset\ C) = Pos\ A$ **and** *neg*: $Neg\ A \in\#\ D$
shows $C\ \#\subset\#\ D$
$\langle proof \rangle$

lemma *pos-Max-imp-neg-notin*: $Max\ (set\text{-}mset\ C) = Pos\ A \Longrightarrow Neg\ A \notin\#\ C$
$\langle proof \rangle$

lemma *less-eq-Max-lit*: $C \neq \{\#\} \Longrightarrow C\ \#\subseteq\#\ D \Longrightarrow Max\ (set\text{-}mset\ C) \leq Max\ (set\text{-}mset\ D)$
$\langle proof \rangle$

definition *atms-of* :: $'a\ clause \Rightarrow 'a\ set$ **where**
$atms\text{-}of\ C = atm\text{-}of\ \text{`}\ set\text{-}mset\ C$

lemma *atms-of-empty*[*simp*]: $atms\text{-}of\ \{\#\} = \{\}$
$\langle proof \rangle$

lemma *atms-of-singleton*[*simp*]: $atms\text{-}of\ \{\#L\#\} = \{atm\text{-}of\ L\}$
$\langle proof \rangle$

lemma *atms-of-union-mset*[*simp*]:
$atms\text{-}of\ (A\ \#\cup\ B) = atms\text{-}of\ A \cup atms\text{-}of\ B$
$\langle proof \rangle$

lemma *finite-atms-of*[*iff*]: $finite\ (atms\text{-}of\ C)$
$\langle proof \rangle$

lemma *atm-of-lit-in-atms-of*: $L \in\#\ C \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\ C$
$\langle proof \rangle$

lemma *atms-of-plus*[*simp*]: $atms\text{-}of\ (C + D) = atms\text{-}of\ C \cup atms\text{-}of\ D$
$\langle proof \rangle$

lemma *pos-lit-in-atms-of*: $Pos\ A \in\#\ C \Longrightarrow A \in atms\text{-}of\ C$
$\langle proof \rangle$

lemma *neg-lit-in-atms-of*: $Neg\ A \in\#\ C \Longrightarrow A \in atms\text{-}of\ C$
$\langle proof \rangle$

lemma *atm-imp-pos-or-neg-lit*: $A \in atms\text{-}of\ C \Longrightarrow Pos\ A \in\#\ C \vee Neg\ A \in\#\ C$
$\langle proof \rangle$

lemma *atm-iff-pos-or-neg-lit*: $A \in atms\text{-}of\ L \longleftrightarrow Pos\ A \in\#\ L \vee Neg\ A \in\#\ L$
$\langle proof \rangle$

lemma *atm-of-eq-atm-of*:
$atm\text{-}of\ L = atm\text{-}of\ L' \longleftrightarrow (L = L' \vee L = -L')$

$\langle proof \rangle$

lemma *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*:
$atm\text{-}of\ L \in atm\text{-}of\ `\ I \longleftrightarrow (L \in I \vee -L \in I)$
$\langle proof \rangle$

lemma *lits-subseteq-imp-atms-subseteq*: $set\text{-}mset\ C \subseteq set\text{-}mset\ D \Longrightarrow atms\text{-}of\ C \subseteq atms\text{-}of\ D$
$\langle proof \rangle$

lemma *atms-empty-iff-empty*[*iff*]: $atms\text{-}of\ C = \{\} \longleftrightarrow C = \{\#\}$
$\langle proof \rangle$

lemma
atms-of-poss[*simp*]: $atms\text{-}of\ (poss\ AA) = set\text{-}mset\ AA$ **and**
atms-of-negg[*simp*]: $atms\text{-}of\ (negs\ AA) = set\text{-}mset\ AA$
$\langle proof \rangle$

lemma *less-eq-Max-atms-of*: $C \neq \{\#\} \Longrightarrow C \subseteq\# D \Longrightarrow Max\ (atms\text{-}of\ C) \leq Max\ (atms\text{-}of\ D)$
$\langle proof \rangle$

lemma *le-multiset-Max-in-imp-Max*:
$Max\ (atms\text{-}of\ D) = A \Longrightarrow C \subseteq\# D \Longrightarrow A \in atms\text{-}of\ C \Longrightarrow Max\ (atms\text{-}of\ C) = A$
$\langle proof \rangle$

lemma *atm-of-Max-lit*[*simp*]: $C \neq \{\#\} \Longrightarrow atm\text{-}of\ (Max\ (set\text{-}mset\ C)) = Max\ (atms\text{-}of\ C)$
$\langle proof \rangle$

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

lemma *atms-less-imp-lit-less-pos*: $(\bigwedge B.\ B \in atms\text{-}of\ C \Longrightarrow B < A) \Longrightarrow L \in\# C \Longrightarrow L < Pos\ A$
$\langle proof \rangle$

lemma *atms-less-eq-imp-lit-less-eq-neg*: $(\bigwedge B.\ B \in atms\text{-}of\ C \Longrightarrow B \leq A) \Longrightarrow L \in\# C \Longrightarrow L \leq Neg\ A$
$\langle proof \rangle$

end

2.2 Herbrand Intepretation

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

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

2.2.1 Herbrand Interpretations

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

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

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

lemma *true-lit-simps*[*simp*]:
$I \models l\ \mathit{Pos}\ A \longleftrightarrow A \in I$
$I \models l\ \mathit{Neg}\ A \longleftrightarrow A \notin I$
⟨*proof*⟩

lemma *true-lit-iff*[*iff*]: $I \models l\ L \longleftrightarrow (\exists A.\ L = \mathit{Pos}\ A \wedge A \in I \vee L = \mathit{Neg}\ A \wedge A \notin I)$
⟨*proof*⟩

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

lemma *true-cls-empty*[*iff*]: $\neg\ I \models \{\#\}$
⟨*proof*⟩

lemma *true-cls-singleton*[*iff*]: $I \models \{\#L\#\} \longleftrightarrow I \models l\ L$
⟨*proof*⟩

lemma *true-cls-union*[*iff*]: $I \models C + D \longleftrightarrow I \models C \vee I \models D$
⟨*proof*⟩

lemma *true-cls-mono*: $\mathit{set\text{-}mset}\ C \subseteq \mathit{set\text{-}mset}\ D \Longrightarrow I \models C \Longrightarrow I \models D$
⟨*proof*⟩

lemma
assumes $I \subseteq J$
shows
false-to-true-imp-ex-pos: $\neg\ I \models C \Longrightarrow J \models C \Longrightarrow \exists A \in J.\ \mathit{Pos}\ A \in\#\ C$ **and**
true-to-false-imp-ex-neg: $I \models C \Longrightarrow \neg\ J \models C \Longrightarrow \exists A \in J.\ \mathit{Neg}\ A \in\#\ C$
⟨*proof*⟩

lemma *true-cls-replicate-mset*[*iff*]: $I \models \mathit{replicate\text{-}mset}\ n\ L \longleftrightarrow n \neq 0 \wedge I \models l\ L$
⟨*proof*⟩

lemma *pos-literal-in-imp-true-cls*[*intro*]: $\mathit{Pos}\ A \in\#\ C \Longrightarrow A \in I \Longrightarrow I \models C$
⟨*proof*⟩

lemma *neg-literal-notin-imp-true-cls*[*intro*]: $\mathit{Neg}\ A \in\#\ C \Longrightarrow A \notin I \Longrightarrow I \models C$
⟨*proof*⟩

lemma *pos-neg-in-imp-true*: $\mathit{Pos}\ A \in\#\ C \Longrightarrow \mathit{Neg}\ A \in\#\ C \Longrightarrow I \models C$
⟨*proof*⟩

definition *true-clss* :: $'a$ *interp* $\Rightarrow$ $'a$ *clause set* $\Rightarrow$ *bool* (**infix** $\models s$ *50*) **where**
$I \models s\ CC \longleftrightarrow (\forall C \in CC.\ I \models C)$

lemma *true-clss-empty*[*iff*]: $I \models s\ \{\}$
⟨*proof*⟩

lemma *true-clss-singleton*[*iff*]: $I \models s\ \{C\} \longleftrightarrow I \models C$
⟨*proof*⟩

lemma *true-clss-union*[*iff*]: $I \models s\ CC \cup DD \longleftrightarrow I \models s\ CC \wedge I \models s\ DD$

⟨*proof*⟩

lemma *true-clss-mono*: $DD \subseteq CC \Longrightarrow I \models s\ CC \Longrightarrow I \models s\ DD$
⟨*proof*⟩

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

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

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

lemma *true-cls-mset-singleton*[*iff*]: $I \models m\ \{\#C\#\} \longleftrightarrow I \models C$
⟨*proof*⟩

lemma *true-cls-mset-union*[*iff*]: $I \models m\ CC + DD \longleftrightarrow I \models m\ CC \wedge I \models m\ DD$
⟨*proof*⟩

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

lemma *true-cls-mset-mono*: $set\text{-}mset\ DD \subseteq set\text{-}mset\ CC \Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
⟨*proof*⟩

lemma *true-clss-set-mset*[*iff*]: $I \models s\ set\text{-}mset\ CC \longleftrightarrow I \models m\ CC$
⟨*proof*⟩

end

2.3 Partial Clausal Logic

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

We define here entailment by a set of literals. This is *not* an Herbrand interpretation and has different properties. One key difference is that such a set can be inconsistent (i.e. containing both L and $-\ L$).

Satisfiability is defined by the existence of a total and consistent model.

2.3.1 Clauses

Clauses are (finite) multisets of literals.

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

2.3.2 Partial Interpretations

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

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

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

Consistency

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

lemma *consistent-interp-empty*[*simp*]:
consistent-interp $\{\}$ ⟨*proof*⟩

lemma *consistent-interp-single*[*simp*]:
consistent-interp $\{L\}$ ⟨*proof*⟩

lemma *consistent-interp-subset*:
assumes
$A \subseteq B$ **and**
consistent-interp B
shows *consistent-interp* A
⟨*proof*⟩

lemma *consistent-interp-change-insert*:
$a \notin A \Longrightarrow -a \notin A \Longrightarrow$ *consistent-interp* (*insert* $(-a)$ A) $\longleftrightarrow$ *consistent-interp* (*insert* a A)
⟨*proof*⟩

lemma *consistent-interp-insert-pos*[*simp*]:
$a \notin A \Longrightarrow$ *consistent-interp* (*insert* a A) $\longleftrightarrow$ *consistent-interp* $A \wedge -a \notin A$
⟨*proof*⟩

lemma *consistent-interp-insert-not-in*:
consistent-interp $A \Longrightarrow a \notin A \Longrightarrow -a \notin A \Longrightarrow$ *consistent-interp* (*insert* a A)
⟨*proof*⟩

Atoms

We define here various lifting of *atm-of* (applied to a single literal) to set and multisets of literals.

definition *atms-of-ms* :: *'a literal multiset set* $\Rightarrow$ *'a set* **where**
atms-of-ms $\psi s = \bigcup$(*atms-of* ‘ ψs)

lemma *atms-of-mmltiset*[*simp*]:
atms-of (*mset* a) = *atm-of* ‘ *set* a
⟨*proof*⟩

lemma *atms-of-ms-mset-unfold*:
atms-of-ms (*mset* ‘ b) = ($\bigcup x \in b.$ *atm-of* ‘ *set* x)
⟨*proof*⟩

definition *atms-of-s* :: *'a literal set* $\Rightarrow$ *'a set* **where**
atms-of-s C = *atm-of* ‘ C

lemma *atms-of-ms-emtpy-set*[*simp*]:
atms-of-ms $\{\} = \{\}$
⟨*proof*⟩

lemma *atms-of-ms-memtpy*[*simp*]:
$atms\text{-}of\text{-}ms\ \{\{\#\}\} = \{\}$
⟨*proof*⟩

lemma *atms-of-ms-mono*:
$A \subseteq B \Longrightarrow atms\text{-}of\text{-}ms\ A \subseteq atms\text{-}of\text{-}ms\ B$
⟨*proof*⟩

lemma *atms-of-ms-finite*[*simp*]:
$finite\ \psi s \Longrightarrow finite\ (atms\text{-}of\text{-}ms\ \psi s)$
⟨*proof*⟩

lemma *atms-of-ms-union*[*simp*]:
$atms\text{-}of\text{-}ms\ (\psi s \cup \chi s) = atms\text{-}of\text{-}ms\ \psi s \cup atms\text{-}of\text{-}ms\ \chi s$
⟨*proof*⟩

lemma *atms-of-ms-insert*[*simp*]:
$atms\text{-}of\text{-}ms\ (insert\ \psi s\ \chi s) = atms\text{-}of\ \psi s \cup atms\text{-}of\text{-}ms\ \chi s$
⟨*proof*⟩

lemma *atms-of-ms-singleton*[*simp*]: $atms\text{-}of\text{-}ms\ \{L\} = atms\text{-}of\ L$
⟨*proof*⟩

lemma *atms-of-atms-of-ms-mono*[*simp*]:
$A \in \psi \Longrightarrow atms\text{-}of\ A \subseteq atms\text{-}of\text{-}ms\ \psi$
⟨*proof*⟩

lemma *atms-of-ms-single-set-mset-atns-of*[*simp*]:
$atms\text{-}of\text{-}ms\ (single\ `\ set\text{-}mset\ B) = atms\text{-}of\ B$
⟨*proof*⟩

lemma *atms-of-ms-remove-incl*:
shows $atms\text{-}of\text{-}ms\ (Set.remove\ a\ \psi) \subseteq atms\text{-}of\text{-}ms\ \psi$
⟨*proof*⟩

lemma *atms-of-ms-remove-subset*:
$atms\text{-}of\text{-}ms\ (\varphi - \psi) \subseteq atms\text{-}of\text{-}ms\ \varphi$
⟨*proof*⟩

lemma *finite-atms-of-ms-remove-subset*[*simp*]:
$finite\ (atms\text{-}of\text{-}ms\ A) \Longrightarrow finite\ (atms\text{-}of\text{-}ms\ (A - C))$
⟨*proof*⟩

lemma *atms-of-ms-empty-iff*:
$atms\text{-}of\text{-}ms\ A = \{\} \longleftrightarrow A = \{\{\#\}\} \lor A = \{\}$
⟨*proof*⟩

lemma *in-implies-atm-of-on-atms-of-ms*:
assumes $L \in\# C$ **and** $C \in N$
shows $atm\text{-}of\ L \in atms\text{-}of\text{-}ms\ N$
⟨*proof*⟩

lemma *in-plus-implies-atm-of-on-atms-of-ms*:
assumes $C+\{\#L\#\} \in N$
shows $atm\text{-}of\ L \in atms\text{-}of\text{-}ms\ N$
⟨*proof*⟩

lemma *in-m-in-literals*:
  assumes $\{\#A\#\} + D \in \psi s$
  shows *atm-of* $A \in$ *atms-of-ms* ψs
  ⟨*proof*⟩

lemma *atms-of-s-union*[*simp*]:
  atms-of-s $(Ia \cup Ib) =$ *atms-of-s* $Ia \cup$ *atms-of-s* Ib
  ⟨*proof*⟩

lemma *atms-of-s-single*[*simp*]:
  atms-of-s $\{L\} = \{$*atm-of* $L\}$
  ⟨*proof*⟩

lemma *atms-of-s-insert*[*simp*]:
  atms-of-s (*insert* L Ib) $= \{$*atm-of* $L\} \cup$ *atms-of-s* Ib
  ⟨*proof*⟩

lemma *in-atms-of-s-decomp*[*iff*]:
  $P \in$ *atms-of-s* $I \longleftrightarrow$ (*Pos* $P \in I \lor$ *Neg* $P \in I$) (**is** *?P* $\longleftrightarrow$ *?Q*)
⟨*proof*⟩

lemma *atm-of-in-atm-of-set-in-uminus*:
  atm-of $L' \in$ *atm-of* ‘ $B \Longrightarrow L' \in B \lor - L' \in B$
  ⟨*proof*⟩

Totality

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

definition *total-over-m* :: *'a literal set* ⇒ *'a clause set* ⇒ *bool* **where**
total-over-m I $\psi s =$ *total-over-set* I (*atms-of-ms* ψs)

lemma *total-over-set-empty*[*simp*]:
  total-over-set I $\{\}$
  ⟨*proof*⟩

lemma *total-over-m-empty*[*simp*]:
  total-over-m I $\{\}$
  ⟨*proof*⟩

lemma *total-over-set-single*[*iff*]:
  total-over-set I $\{L\} \longleftrightarrow$ (*Pos* $L \in I \lor$ *Neg* $L \in I$)
  ⟨*proof*⟩

lemma *total-over-set-insert*[*iff*]:
  total-over-set I (*insert* L Ls) $\longleftrightarrow$ ((*Pos* $L \in I \lor$ *Neg* $L \in I$) $\land$ *total-over-set* I Ls)
  ⟨*proof*⟩

lemma *total-over-set-union*[*iff*]:
  total-over-set I $(Ls \cup Ls') \longleftrightarrow$ (*total-over-set* I $Ls \land$ *total-over-set* I Ls')
  ⟨*proof*⟩

lemma *total-over-m-subset*:
  $A \subseteq B \Longrightarrow$ *total-over-m* I $B \Longrightarrow$ *total-over-m* I A

$\langle proof \rangle$

lemma *total-over-m-sum*[*iff*]:
shows *total-over-m I* $\{C + D\} \longleftrightarrow$ (*total-over-m I* $\{C\} \wedge$ *total-over-m I* $\{D\}$)
$\langle proof \rangle$

lemma *total-over-m-union*[*iff*]:
total-over-m I $(A \cup B) \longleftrightarrow$ (*total-over-m I A* $\wedge$ *total-over-m I B*)
$\langle proof \rangle$

lemma *total-over-m-insert*[*iff*]:
total-over-m I (*insert a A*) $\longleftrightarrow$ (*total-over-set I* (*atms-of a*) $\wedge$ *total-over-m I A*)
$\langle proof \rangle$

lemma *total-over-m-extension*:
fixes I :: *'v literal set* **and** A :: *'v clauses*
assumes *total*: *total-over-m I A*
shows $\exists I'$. *total-over-m* $(I \cup I')$ $(A \cup B)$
$\wedge$ ($\forall x \in I'$. *atm-of x* $\in$ *atms-of-ms B* $\wedge$ *atm-of x* $\notin$ *atms-of-ms A*)
$\langle proof \rangle$

lemma *total-over-m-consistent-extension*:
fixes I :: *'v literal set* **and** A :: *'v clauses*
assumes
total: *total-over-m I A* **and**
cons: *consistent-interp I*
shows $\exists I'$. *total-over-m* $(I \cup I')$ $(A \cup B)$
$\wedge$ ($\forall x \in I'$. *atm-of x* $\in$ *atms-of-ms B* $\wedge$ *atm-of x* $\notin$ *atms-of-ms A*) $\wedge$ *consistent-interp* $(I \cup I')$
$\langle proof \rangle$

lemma *total-over-set-atms-of-m*[*simp*]:
total-over-set Ia (*atms-of-s Ia*)
$\langle proof \rangle$

lemma *total-over-set-literal-defined*:
assumes $\{\#A\#\} + D \in \psi s$
and *total-over-set I* (*atms-of-ms* ψs)
shows $A \in I \vee -A \in I$
$\langle proof \rangle$

lemma *tot-over-m-remove*:
assumes *total-over-m* $(I \cup \{L\})$ $\{\psi\}$
and L: $L \notin\# \psi$ $-L \notin\# \psi$
shows *total-over-m I* $\{\psi\}$
$\langle proof \rangle$

lemma *total-union*:
assumes *total-over-m I* ψ
shows *total-over-m* $(I \cup I')$ ψ
$\langle proof \rangle$

lemma *total-union-2*:
assumes *total-over-m I* ψ
and *total-over-m* I' ψ'
shows *total-over-m* $(I \cup I')$ $(\psi \cup \psi')$
$\langle proof \rangle$

Interpretations

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

lemma *true-cls-empty*[*iff*]: $\neg I \models \{\#\}$
  ⟨*proof*⟩

lemma *true-cls-singleton*[*iff*]: $I \models \{\#L\#\} \longleftrightarrow I \models l\ L$
  ⟨*proof*⟩

lemma *true-cls-union*[*iff*]: $I \models C + D \longleftrightarrow I \models C \lor I \models D$
  ⟨*proof*⟩

lemma *true-cls-mono-set-mset*: *set-mset* $C \subseteq$ *set-mset* $D \Longrightarrow I \models C \Longrightarrow I \models D$
  ⟨*proof*⟩

lemma *true-cls-mono-leD*[*dest*]: $A \subseteq\# B \Longrightarrow I \models A \Longrightarrow I \models B$
  ⟨*proof*⟩

lemma
  assumes $I \models \psi$
  shows
    true-cls-union-increase[*simp*]: $I \cup I' \models \psi$ **and**
    true-cls-union-increase'[*simp*]: $I' \cup I \models \psi$
  ⟨*proof*⟩

lemma *true-cls-mono-set-mset-l*:
  assumes $A \models \psi$
  and $A \subseteq B$
  shows $B \models \psi$
  ⟨*proof*⟩

lemma *true-cls-replicate-mset*[*iff*]: $I \models$ *replicate-mset* $n\ L \longleftrightarrow n \neq 0 \land I \models l\ L$
  ⟨*proof*⟩

lemma *true-cls-empty-entails*[*iff*]: $\neg \{\} \models N$
  ⟨*proof*⟩

lemma *true-cls-not-in-remove*:
  assumes $L \notin\# \chi$ **and** $I \cup \{L\} \models \chi$
  shows $I \models \chi$
  ⟨*proof*⟩

definition *true-clss* :: $'a$ *interp* $\Rightarrow$ $'a$ *clauses* $\Rightarrow$ *bool* (**infix** $\models s$ *50*) **where**
  $I \models s\ CC \longleftrightarrow (\forall C \in CC. I \models C)$

lemma *true-clss-empty*[*simp*]: $I \models s\ \{\}$
  ⟨*proof*⟩

lemma *true-clss-singleton*[*iff*]: $I \models s\ \{C\} \longleftrightarrow I \models C$
  ⟨*proof*⟩

lemma *true-clss-empty-entails-empty*[*iff*]: $\{\} \models s\ N \longleftrightarrow N = \{\}$
  ⟨*proof*⟩

lemma *true-cls-insert-l* [*simp*]:
$M \models A \Longrightarrow insert\ L\ M \models A$
⟨*proof*⟩

lemma *true-clss-union*[*iff*]: $I \models s\ CC \cup DD \longleftrightarrow I \models s\ CC \wedge I \models s\ DD$
⟨*proof*⟩

lemma *true-clss-insert*[*iff*]: $I \models s\ insert\ C\ DD \longleftrightarrow I \models C \wedge I \models s\ DD$
⟨*proof*⟩

lemma *true-clss-mono*: $DD \subseteq CC \Longrightarrow I \models s\ CC \Longrightarrow I \models s\ DD$
⟨*proof*⟩

lemma *true-clss-union-increase*[*simp*]:
assumes $I \models s\ \psi$
shows $I \cup I' \models s\ \psi$
⟨*proof*⟩

lemma *true-clss-union-increase'*[*simp*]:
assumes $I' \models s\ \psi$
shows $I \cup I' \models s\ \psi$
⟨*proof*⟩

lemma *true-clss-commute-l*:
$(I \cup I' \models s\ \psi) \longleftrightarrow (I' \cup I \models s\ \psi)$
⟨*proof*⟩

lemma *model-remove*[*simp*]: $I \models s\ N \Longrightarrow I \models s\ Set.remove\ a\ N$
⟨*proof*⟩

lemma *model-remove-minus*[*simp*]: $I \models s\ N \Longrightarrow I \models s\ N - A$
⟨*proof*⟩

lemma *notin-vars-union-true-cls-true-cls*:
assumes $\forall x \in I'.\ atm\text{-}of\ x \notin atms\text{-}of\text{-}ms\ A$
and $atms\text{-}of\ L \subseteq atms\text{-}of\text{-}ms\ A$
and $I \cup I' \models L$
shows $I \models L$
⟨*proof*⟩

lemma *notin-vars-union-true-clss-true-clss*:
assumes $\forall x \in I'.\ atm\text{-}of\ x \notin atms\text{-}of\text{-}ms\ A$
and $atms\text{-}of\text{-}ms\ L \subseteq atms\text{-}of\text{-}ms\ A$
and $I \cup I' \models s\ L$
shows $I \models s\ L$
⟨*proof*⟩

Satisfiability

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

lemma *satisfiable-single*[*simp*]:
$satisfiable\ \{\{\#L\#\}\}$
⟨*proof*⟩

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

lemma *satisfiable-decreasing*:
assumes *satisfiable* $(\psi \cup \psi')$
shows *satisfiable* ψ
⟨*proof*⟩

lemma *satisfiable-def-min*:
satisfiable CC
$\longleftrightarrow$ $(\exists I.\ I \models s\ CC \wedge consistent\text{-}interp\ I \wedge total\text{-}over\text{-}m\ I\ CC \wedge atm\text{-}of`I = atms\text{-}of\text{-}ms\ CC)$
(**is** *?sat* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *satisfiable-carac*[*iff*]:
$(\exists I.\ consistent\text{-}interp\ I \wedge I \models s\ \varphi) \longleftrightarrow satisfiable\ \varphi$ (**is** $(\exists I.\ ?Q\ I) \longleftrightarrow ?S$)
⟨*proof*⟩

lemma *satisfiable-carac'*[*simp*]: $consistent\text{-}interp\ I \Longrightarrow I \models s\ \varphi \Longrightarrow satisfiable\ \varphi$
⟨*proof*⟩

Entailment for Multisets of Clauses

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

lemma *true-cls-mset-empty*[*simp*]: $I \models m\ \{\#\}$
⟨*proof*⟩

lemma *true-cls-mset-singleton*[*iff*]: $I \models m\ \{\#C\#\} \longleftrightarrow I \models C$
⟨*proof*⟩

lemma *true-cls-mset-union*[*iff*]: $I \models m\ CC + DD \longleftrightarrow I \models m\ CC \wedge I \models m\ DD$
⟨*proof*⟩

lemma *true-cls-mset-image-mset*[*iff*]: $I \models m\ image\text{-}mset\ f\ A \longleftrightarrow (\forall x \in\#\ A.\ I \models f\ x)$
⟨*proof*⟩

lemma *true-cls-mset-mono*: $set\text{-}mset\ DD \subseteq set\text{-}mset\ CC \Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
⟨*proof*⟩

lemma *true-clss-set-mset*[*iff*]: $I \models s\ set\text{-}mset\ CC \longleftrightarrow I \models m\ CC$
⟨*proof*⟩

lemma *true-cls-mset-increasing-r*[*simp*]:
$I \models m\ CC \Longrightarrow I \cup J \models m\ CC$
⟨*proof*⟩

theorem *true-cls-remove-unused*:
assumes $I \models \psi$
shows $\{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\ \psi\} \models \psi$
⟨*proof*⟩

theorem *true-clss-remove-unused*:
assumes $I \models s\ \psi$
shows $\{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\} \models s\ \psi$

⟨*proof*⟩

A simple application of the previous theorem:

lemma *true-clss-union-decrease*:
assumes II': $I \cup I' \models \psi$
and H: $\forall v \in I'. \ atm\text{-}of\ v \notin atms\text{-}of\ \psi$
shows $I \models \psi$
⟨*proof*⟩

lemma *multiset-not-empty*:
assumes $M \neq \{\#\}$
and $x \in\# M$
shows $\exists A.\ x = Pos\ A \lor x = Neg\ A$
⟨*proof*⟩

lemma *atms-of-ms-empty*:
fixes $\psi :: 'v\ clauses$
assumes $atms\text{-}of\text{-}ms\ \psi = \{\}$
shows $\psi = \{\} \lor \psi = \{\{\#\}\}$
⟨*proof*⟩

lemma *consistent-interp-disjoint*:
assumes *consI*: *consistent-interp I*
and *disj*: $atms\text{-}of\text{-}s\ A \cap atms\text{-}of\text{-}s\ I = \{\}$
and *consA*: *consistent-interp A*
shows *consistent-interp* $(A \cup I)$
⟨*proof*⟩

lemma *total-remove-unused*:
assumes $total\text{-}over\text{-}m\ I\ \psi$
shows $total\text{-}over\text{-}m\ \{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\}\ \psi$
⟨*proof*⟩

lemma *true-cls-remove-hd-if-notin-vars*:
assumes $insert\ a\ M' \models D$
and $atm\text{-}of\ a \notin atms\text{-}of\ D$
shows $M' \models D$
⟨*proof*⟩

lemma *total-over-set-atm-of*:
fixes $I :: 'v\ interp$ **and** $K :: 'v\ set$
shows $total\text{-}over\text{-}set\ I\ K \longleftrightarrow (\forall l \in K.\ l \in (atm\text{-}of\ `\ I))$
⟨*proof*⟩

Tautologies

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

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

lemma *tautology-Pos-Neg*[*intro*]:
assumes $Pos\ p \in\# A$ **and** $Neg\ p \in\# A$
shows *tautology A*
⟨*proof*⟩

lemma *tautology-minus*[*simp*]:
  assumes $L \in\# A$ **and** $-L \in\# A$
  shows *tautology A*
  ⟨*proof*⟩

lemma *tautology-exists-Pos-Neg*:
  assumes *tautology* ψ
  shows $\exists p.\ Pos\ p \in\# \psi \wedge Neg\ p \in\# \psi$
⟨*proof*⟩

lemma *tautology-decomp*:
  $tautology\ \psi \longleftrightarrow (\exists p.\ Pos\ p \in\# \psi \wedge Neg\ p \in\# \psi)$
  ⟨*proof*⟩

lemma *tautology-false*[*simp*]: $\neg tautology\ \{\#\}$
  ⟨*proof*⟩

lemma *tautology-add-single*:
  $tautology\ (\{\#a\#\} + L) \longleftrightarrow tautology\ L \vee -a \in\# L$
  ⟨*proof*⟩

lemma *minus-interp-tautology*:
  assumes $\{-L \mid L.\ L\in\# \chi\} \models \chi$
  shows *tautology* χ
⟨*proof*⟩

lemma *remove-literal-in-model-tautology*:
  assumes $I \cup \{Pos\ P\} \models \varphi$
  and $I \cup \{Neg\ P\} \models \varphi$
  shows $I \models \varphi \vee tautology\ \varphi$
  ⟨*proof*⟩

lemma *tautology-imp-tautology*:
  fixes $\chi\ \chi' :: 'v\ clause$
  assumes $\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$ **and** *tautology* χ
  shows *tautology* χ' ⟨*proof*⟩

Entailment for clauses and propositions

We also need entailment of clauses by other clauses.

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

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

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

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

lemma *true-cls-cls-refl*[*simp*]:
  $A \models f\ A$
  ⟨*proof*⟩

lemma *true-cls-cls-insert-l*[*simp*]:
$a \models f\ C \Longrightarrow insert\ a\ A \models p\ C$
⟨*proof*⟩

lemma *true-cls-clss-empty*[*iff*]:
$N \models fs\ \{\}$
⟨*proof*⟩

lemma *true-prop-true-clause*[*iff*]:
$\{\varphi\} \models p\ \psi \longleftrightarrow \varphi \models f\ \psi$
⟨*proof*⟩

lemma *true-clss-clss-true-clss-cls*[*iff*]:
$N \models ps\ \{\psi\} \longleftrightarrow N \models p\ \psi$
⟨*proof*⟩

lemma *true-clss-clss-true-cls-clss*[*iff*]:
$\{\chi\} \models ps\ \psi \longleftrightarrow \chi \models fs\ \psi$
⟨*proof*⟩

lemma *true-clss-clss-empty*[*simp*]:
$N \models ps\ \{\}$
⟨*proof*⟩

lemma *true-clss-cls-subset*:
$A \subseteq B \Longrightarrow A \models p\ CC \Longrightarrow B \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cs-mono-l*[*simp*]:
$A \models p\ CC \Longrightarrow A \cup B \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cs-mono-l2*[*simp*]:
$B \models p\ CC \Longrightarrow A \cup B \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cls-mono-r*[*simp*]:
$A \models p\ CC \Longrightarrow A \models p\ CC + CC'$
⟨*proof*⟩

lemma *true-clss-cls-mono-r'*[*simp*]:
$A \models p\ CC' \Longrightarrow A \models p\ CC + CC'$
⟨*proof*⟩

lemma *true-clss-clss-union-l*[*simp*]:
$A \models ps\ CC \Longrightarrow A \cup B \models ps\ CC$
⟨*proof*⟩

lemma *true-clss-clss-union-l-r*[*simp*]:
$B \models ps\ CC \Longrightarrow A \cup B \models ps\ CC$
⟨*proof*⟩

lemma *true-clss-cls-in*[*simp*]:
$CC \in A \Longrightarrow A \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cls-insert-l*[*simp*]:
$A \models p\ C \Longrightarrow insert\ a\ A \models p\ C$
⟨*proof*⟩

lemma *true-clss-clss-insert-l*[*simp*]:
$A \models ps\ C \Longrightarrow insert\ a\ A \models ps\ C$
⟨*proof*⟩

lemma *true-clss-clss-union-and*[*iff*]:
$A \models ps\ C \cup D \longleftrightarrow (A \models ps\ C \wedge A \models ps\ D)$
⟨*proof*⟩

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

lemma *true-clss-clss-subset*:
$A \subseteq B \Longrightarrow A \models ps\ CC \Longrightarrow B \models ps\ CC$
⟨*proof*⟩

lemma *union-trus-clss-clss*[*simp*]: $A \cup B \models ps\ B$
⟨*proof*⟩

lemma *true-clss-clss-remove*[*simp*]:
$A \models ps\ B \Longrightarrow A \models ps\ B - C$
⟨*proof*⟩

lemma *true-clss-clss-subsetE*:
$N \models ps\ B \Longrightarrow A \subseteq B \Longrightarrow N \models ps\ A$
⟨*proof*⟩

lemma *true-clss-clss-in-imp-true-clss-cls*:
assumes $N \models ps\ U$
and $A \in U$
shows $N \models p\ A$
⟨*proof*⟩

lemma *all-in-true-clss-clss*: $\forall x \in B.\ x \in A \Longrightarrow A \models ps\ B$
⟨*proof*⟩

lemma *true-clss-clss-left-right*:
assumes $A \models ps\ B$
and $A \cup B \models ps\ M$
shows $A \models ps\ M \cup B$
⟨*proof*⟩

lemma *true-clss-clss-generalise-true-clss-clss*:
$A \cup C \models ps\ D \Longrightarrow B \models ps\ C \Longrightarrow A \cup B \models ps\ D$
⟨*proof*⟩

lemma *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*:
assumes D: $N \models p\ D + \{\#- L\#\}$
and C: $N \models p\ C + \{\#L\#\}$
shows $N \models p\ D + C$
⟨*proof*⟩

lemma *true-cls-union-mset*[*iff*]: $I \models C \ \#\cup \ D \longleftrightarrow I \models C \lor I \models D$
⟨*proof*⟩

lemma *true-clss-cls-union-mset-true-clss-cls-or-not-true-clss-cls-or*:
assumes
D: $N \models p \ D + \{\#- \ L\#\}$ **and**
C: $N \models p \ C + \{\#L\#\}$
shows $N \models p \ D \ \#\cup \ C$
⟨*proof*⟩

2.3.3 Subsumptions

lemma *subsumption-total-over-m*:
assumes $A \subseteq\# \ B$
shows *total-over-m* I $\{B\} \Longrightarrow$ *total-over-m* I $\{A\}$
⟨*proof*⟩

lemma *atms-of-replicate-mset-replicate-mset-uminus*[*simp*]:
atms-of (D − *replicate-mset* (*count* D L) L − *replicate-mset* (*count* D ($-L$)) ($-L$))
= *atms-of* D − {*atm-of* L}
⟨*proof*⟩

lemma *subsumption-chained*:
assumes
$\forall I.$ *total-over-m* I $\{D\} \longrightarrow I \models D \longrightarrow I \models \varphi$ **and**
$C \subseteq\# \ D$
shows ($\forall I.$ *total-over-m* I $\{C\} \longrightarrow I \models C \longrightarrow I \models \varphi$) $\lor$ *tautology* φ
⟨*proof*⟩

2.3.4 Removing Duplicates

lemma *tautology-remdups-mset*[*iff*]:
tautology (*remdups-mset* C) $\longleftrightarrow$ *tautology* C
⟨*proof*⟩

lemma *atms-of-remdups-mset*[*simp*]: *atms-of* (*remdups-mset* C) = *atms-of* C
⟨*proof*⟩

lemma *true-cls-remdups-mset*[*iff*]: $I \models$ *remdups-mset* $C \longleftrightarrow I \models C$
⟨*proof*⟩

lemma *true-clss-cls-remdups-mset*[*iff*]: $A \models p$ *remdups-mset* $C \longleftrightarrow A \models p \ C$
⟨*proof*⟩

2.3.5 Set of all Simple Clauses

A simple clause with respect to a set of atoms is such that

1. its atoms are included in the considered set of atoms;
2. it is not a tautology;
3. it does not contains duplicate literals.

   It corresponds to the clauses that cannot be simplified away in a calculus without considering the other clauses.

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

lemma *simple-clss-empty*[*simp*]:
  simple-clss {} = {{#}}
  ⟨*proof*⟩

lemma *simple-clss-insert*:
  assumes $l \notin$ *atms*
  shows *simple-clss* (*insert l atms*) =
    (*op* + {#*Pos l*#}) ' (*simple-clss atms*)
    ∪ (*op* + {#*Neg l*#}) ' (*simple-clss atms*)
    ∪ *simple-clss atms*(**is** *?I* = *?U*)
⟨*proof*⟩

lemma *simple-clss-finite*:
  fixes *atms* :: *'v set*
  assumes *finite atms*
  shows *finite* (*simple-clss atms*)
  ⟨*proof*⟩

lemma *simple-clssE*:
  assumes
    $x \in$ *simple-clss atms*
  shows *atms-of* $x \subseteq$ *atms* $\wedge \neg$*tautology* $x \wedge$ *distinct-mset* x
  ⟨*proof*⟩

lemma *cls-in-simple-clss*:
  shows {#} ∈ *simple-clss s*
  ⟨*proof*⟩

lemma *simple-clss-card*:
  fixes *atms* :: *'v set*
  assumes *finite atms*
  shows *card* (*simple-clss atms*) ≤ (*3*::*nat*) ^ (*card atms*)
  ⟨*proof*⟩

lemma *simple-clss-mono*:
  assumes *incl*: *atms* ⊆ *atms'*
  shows *simple-clss atms* ⊆ *simple-clss atms'*
  ⟨*proof*⟩

lemma *distinct-mset-not-tautology-implies-in-simple-clss*:
  assumes *distinct-mset* χ **and** $\neg$*tautology* χ
  shows $\chi \in$ *simple-clss* (*atms-of* χ)
  ⟨*proof*⟩

lemma *simplified-in-simple-clss*:
  assumes *distinct-mset-set* ψ **and** $\forall \chi \in \psi.\ \neg$*tautology* χ
  shows $\psi \subseteq$ *simple-clss* (*atms-of-ms* ψ)
  ⟨*proof*⟩

2.3.6 Experiment: Expressing the Entailments as Locales

locale *entail* =
  fixes *entail* :: *'a set* ⇒ *'b* ⇒ *bool* (**infix** ⊨*e 50*)

assumes *entail-insert*[*simp*]: $I \neq \{\} \Longrightarrow$ *insert* $L\ I \models e\ x \longleftrightarrow \{L\} \models e\ x \lor I \models e\ x$
assumes *entail-union*[*simp*]: $I \models e\ A \Longrightarrow I \cup I' \models e\ A$
begin

definition *entails* :: $'a$ *set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ *bool* (**infix** $\models$*es 50*) **where**
$I \models es\ A \longleftrightarrow (\forall a \in A.\ I \models e\ a)$

lemma *entails-empty*[*simp*]:
$I \models es\ \{\}$
⟨*proof*⟩

lemma *entails-single*[*iff*]:
$I \models es\ \{a\} \longleftrightarrow I \models e\ a$
⟨*proof*⟩

lemma *entails-insert-l*[*simp*]:
$M \models es\ A \Longrightarrow$ *insert* $L\ M \models es\ A$
⟨*proof*⟩

lemma *entails-union*[*iff*]: $I \models es\ CC \cup DD \longleftrightarrow I \models es\ CC \land I \models es\ DD$
⟨*proof*⟩

lemma *entails-insert*[*iff*]: $I \models es$ *insert* $C\ DD \longleftrightarrow I \models e\ C \land I \models es\ DD$
⟨*proof*⟩

lemma *entails-insert-mono*: $DD \subseteq CC \Longrightarrow I \models es\ CC \Longrightarrow I \models es\ DD$
⟨*proof*⟩

lemma *entails-union-increase*[*simp*]:
assumes $I \models es\ \psi$
shows $I \cup I' \models es\ \psi$
⟨*proof*⟩

lemma *true-clss-commute-l*:
$I \cup I' \models es\ \psi \longleftrightarrow I' \cup I \models es\ \psi$
⟨*proof*⟩

lemma *entails-remove*[*simp*]: $I \models es\ N \Longrightarrow I \models es$ *Set.remove* $a\ N$
⟨*proof*⟩

lemma *entails-remove-minus*[*simp*]: $I \models es\ N \Longrightarrow I \models es\ N - A$
⟨*proof*⟩

end

interpretation *true-cls*: *entail true-cls*
⟨*proof*⟩

2.3.7 Entailment to be extended

In some cases we want a more general version of entailment to have for example $\{\} \models \{\#L, -L\#\}$. This is useful when the model we are building might not be total (the literal L might have been definitely removed from the set of clauses), but we still want to have a property of entailment considering that theses removed literals are not important.

We can given a model I consider all the natural extensions: C is entailed by an extended I, if for all total extension of I, this model entails C.

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

lemma *true-clss-imp-true-cls-ext*:
  I⊨*s* $N \Longrightarrow I$ ⊨*sext* N
  ⟨*proof*⟩

lemma *true-clss-ext-decrease-right-remove-r*:
  assumes I ⊨*sext* N
  shows I ⊨*sext* $N - \{C\}$
  ⟨*proof*⟩

lemma *consistent-true-clss-ext-satisfiable*:
  assumes *consistent-interp* I **and** I ⊨*sext* A
  shows *satisfiable* A
  ⟨*proof*⟩

lemma *not-consistent-true-clss-ext*:
  assumes ¬*consistent-interp* I
  shows I ⊨*sext* A
  ⟨*proof*⟩
end
theory *Prop-Logic*
imports *Main*
begin

Chapter 3

Normalisation

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

3.1 Logics

In this section we define the syntax of the formula and an abstraction over it to have simpler proofs. After that we define some properties like subformula and rewriting.

3.1.1 Definition and abstraction

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

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

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

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

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

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

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

lemma *propo-induct-arity*[*case-names nullary unary binary*]:
fixes $\varphi\ \psi :: 'v\ propo$
assumes *nullary*: $\bigwedge\varphi\ x.\ \varphi = FF \vee \varphi = FT \vee \varphi = FVar\ x \Longrightarrow P\ \varphi$
and *unary*: $\bigwedge\psi.\ P\ \psi \Longrightarrow P\ (FNot\ \psi)$
and *binary*: $\bigwedge\varphi\ \psi 1\ \psi 2.\ P\ \psi 1 \Longrightarrow P\ \psi 2 \Longrightarrow \varphi = FAnd\ \psi 1\ \psi 2 \vee \varphi = FOr\ \psi 1\ \psi 2 \vee \varphi = FImp\ \psi 1$
$\psi 2$
$\vee\ \varphi = FEq\ \psi 1\ \psi 2 \Longrightarrow P\ \varphi$
shows $P\ \psi$
$\langle proof \rangle$

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

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

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

lemma *connective-cases-arity*[*case-names nullary binary unary*]:
  assumes *nullary*: $\bigwedge x.\ c = CT \vee c = CF \vee c = CVar\ x \Longrightarrow P$
  and *binary*: $c \in$ *binary-connectives* $\Longrightarrow P$
  and *unary*: $c = CNot \Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

lemma *connective-cases-arity-2*[*case-names nullary unary binary*]:
  assumes *nullary*: $c \in$ *nullary-connective* $\Longrightarrow P$
  and *unary*: $c = CNot \Longrightarrow P$
  and *binary*: $c \in$ *binary-connectives* $\Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

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

inductive *wf-conn* :: *'v connective* ⇒ *'v propo list* ⇒ *bool* **for** *c* :: *'v connective* **where**
wf-conn-nullary[*simp*]: ($c = CT \vee c = CF \vee c = CVar\ v$) $\Longrightarrow$ *wf-conn c* [] |
wf-conn-unary[*simp*]: $c = CNot \Longrightarrow$ *wf-conn c* [ψ] |
wf-conn-binary[*simp*]: $c \in$ *binary-connectives* $\Longrightarrow$ *wf-conn c* (ψ # ψ' # [])
thm *wf-conn.induct*
lemma *wf-conn-induct*[*consumes 1*, *case-names CT CF CVar CNot COr CAnd CImp CEq*]:
  assumes *wf-conn c x* **and**
    $\bigwedge v.\ c = CT \Longrightarrow P$ [] **and**
    $\bigwedge v.\ c = CF \Longrightarrow P$ [] **and**
    $\bigwedge v.\ c = CVar\ v \Longrightarrow P$ [] **and**
    $\bigwedge \psi.\ c = CNot \Longrightarrow P$ [ψ] **and**
    $\bigwedge \psi\ \psi'.\ c = COr \Longrightarrow P$ [ψ, ψ'] **and**
    $\bigwedge \psi\ \psi'.\ c = CAnd \Longrightarrow P$ [ψ, ψ'] **and**
    $\bigwedge \psi\ \psi'.\ c = CImp \Longrightarrow P$ [ψ, ψ'] **and**
    $\bigwedge \psi\ \psi'.\ c = CEq \Longrightarrow P$ [ψ, ψ']
  shows *P x*
  ⟨*proof*⟩

3.1.2 properties of the abstraction

First we can define simplification rules.

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

$wf\text{-}conn\ CT\ l \Longrightarrow conn\ CT\ l = FT$
$wf\text{-}conn\ CF\ l \Longrightarrow conn\ CF\ l = FF$
$wf\text{-}conn\ (CVar\ x)\ l \Longrightarrow conn\ (CVar\ x)\ l = FVar\ x$
⟨*proof*⟩

lemma *wf-conn-list-decomp*[*simp*]:
$wf\text{-}conn\ CT\ l \longleftrightarrow l = []$
$wf\text{-}conn\ CF\ l \longleftrightarrow l = []$
$wf\text{-}conn\ (CVar\ x)\ l \longleftrightarrow l = []$
$wf\text{-}conn\ CNot\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow \xi = [] \wedge \xi' = []$
⟨*proof*⟩

lemma *wf-conn-list*:
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FT \longleftrightarrow (c = CT \wedge l = [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FF \longleftrightarrow (c = CF \wedge l = [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FVar\ x \longleftrightarrow (c = CVar\ x \wedge l = [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FAnd\ a\ b \longleftrightarrow (c = CAnd \wedge l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FOr\ a\ b \longleftrightarrow (c = COr \wedge l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FEq\ a\ b \longleftrightarrow (c = CEq \wedge l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FImp\ a\ b \longleftrightarrow (c = CImp \wedge l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FNot\ a \longleftrightarrow (c = CNot \wedge l = a\ \#\ [])$
⟨*proof*⟩

In the binary connective cases, we will often decompose the list of arguments (of length 2) into two elements.

lemma *list-length2-decomp*: $length\ l = 2 \Longrightarrow (\exists\ a\ b.\ l = a\ \#\ b\ \#\ [])$
⟨*proof*⟩

wf-conn for binary operators means that there are two arguments.

lemma *wf-conn-bin-list-length*:
fixes $l :: 'v\ propo\ list$
assumes *conn*: $c \in binary\text{-}connectives$
shows $length\ l = 2 \longleftrightarrow wf\text{-}conn\ c\ l$
⟨*proof*⟩

lemma *wf-conn-not-list-length*[*iff*]:
fixes $l :: 'v\ propo\ list$
shows $wf\text{-}conn\ CNot\ l \longleftrightarrow length\ l = 1$
⟨*proof*⟩

Decomposing the Not into an element is moreover very useful.

lemma *wf-conn-Not-decomp*:
fixes $l :: 'v\ propo\ list$ **and** $a :: 'v$
assumes *corr*: $wf\text{-}conn\ CNot\ l$
shows $\exists\ a.\ l = [a]$
⟨*proof*⟩

The *wf-conn* remains correct if the length of list does not change. This lemma is very useful when we do one rewriting step

lemma *wf-conn-no-arity-change*:
$length\ l = length\ l' \Longrightarrow wf\text{-}conn\ c\ l \longleftrightarrow wf\text{-}conn\ c\ l'$
⟨*proof*⟩

lemma *wf-conn-no-arity-change-helper*:
  $length\ (\xi\ @\ \varphi\ \#\ \xi') = length\ (\xi\ @\ \varphi'\ \#\ \xi')$
  ⟨*proof*⟩

The injectivity of *conn* is useful to prove equality of the connectives and the lists.

lemma *conn-inj-not*:
  assumes *correct*: *wf-conn c l*
  and *conn*: $conn\ c\ l = FNot\ \psi$
  shows $c = CNot$ **and** $l = [\psi]$
  ⟨*proof*⟩

lemma *conn-inj*:
  fixes $c\ ca$ :: *'v connective* **and** $l\ \psi s$ :: *'v propo list*
  assumes *corr*: *wf-conn ca l*
  and *corr'*: $wf\text{-}conn\ c\ \psi s$
  and *eq*: $conn\ ca\ l = conn\ c\ \psi s$
  shows $ca = c \wedge \psi s = l$
  ⟨*proof*⟩

3.1.3 Subformulas and properties

A characterization using sub-formulas is interesting for rewriting: we will define our relation on the sub-term level, and then lift the rewriting on the term-level. So the rewriting takes place on a subformula.

inductive *subformula* :: *'v propo* ⇒ *'v propo* ⇒ *bool* (**infix** $\preceq$ *45*) **for** φ **where**
subformula-refl[*simp*]: $\varphi \preceq \varphi$ |
subformula-into-subformula: $\psi \in set\ l \Longrightarrow wf\text{-}conn\ c\ l \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq conn\ c\ l$

On the *subformula-into-subformula*, we can see why we use our *conn* representation: one case is enough to express the subformulas property instead of listing all the cases.

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:
shows *b*: $FNot\ \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi$
  ⟨*proof*⟩

lemma *subformula-in-binary-conn*:
  assumes *conn*: $c \in binary\text{-}connectives$
  shows $f \preceq conn\ c\ [f, g]$
  and $g \preceq conn\ c\ [f, g]$
⟨*proof*⟩

lemma *subformula-trans*:
 $\psi \preceq \psi' \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi'$
  ⟨*proof*⟩

lemma *subformula-leaf*:
  fixes $\varphi\ \psi$ :: *'v propo*
  assumes *incl*: $\varphi \preceq \psi$
  and *simple*: $\psi = FT \vee \psi = FF \vee \psi = FVar\ x$
  shows $\varphi = \psi$
  ⟨*proof*⟩

lemma *subfurmula-not-incl-eq*:

assumes $\varphi \preceq$ *conn c l*
and *wf-conn c l*
and $\forall \psi.\ \psi \in$ *set l* $\longrightarrow \neg\ \varphi \preceq \psi$
shows $\varphi =$ *conn c l*
⟨*proof*⟩

lemma *wf-subformula-conn-cases*:
wf-conn c l $\implies \varphi \preceq$ *conn c l* $\longleftrightarrow (\varphi =$ *conn c l* $\vee (\exists \psi.\ \psi \in$ *set l* $\wedge\ \varphi \preceq \psi))$
⟨*proof*⟩

lemma *subformula-decomp-explicit*[*simp*]:
$\varphi \preceq$ *FAnd* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FAnd* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$ (**is** *?P FAnd*)
$\varphi \preceq$ *FOr* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FOr* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\varphi \preceq$ *FEq* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FEq* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\varphi \preceq$ *FImp* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FImp* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
⟨*proof*⟩

lemma *wf-conn-helper-facts*[*iff*]:
wf-conn CNot $[\varphi]$
wf-conn CT []
wf-conn CF []
wf-conn (*CVar x*) []
wf-conn CAnd $[\varphi, \psi]$
wf-conn COr $[\varphi, \psi]$
wf-conn CImp $[\varphi, \psi]$
wf-conn CEq $[\varphi, \psi]$
⟨*proof*⟩

lemma *exists-c-conn*: $\exists$ *c l*. $\varphi =$ *conn c l* $\wedge$ *wf-conn c l*
⟨*proof*⟩

lemma *subformula-conn-decomp*[*simp*]:
assumes *wf*: *wf-conn c l*
shows $\varphi \preceq$ *conn c l* $\longleftrightarrow (\varphi =$ *conn c l* $\vee (\exists\ \psi \in$ *set l*. $\varphi \preceq \psi))$ (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *subformula-leaf-explicit*[*simp*]:
$\varphi \preceq$ *FT* $\longleftrightarrow \varphi =$ *FT*
$\varphi \preceq$ *FF* $\longleftrightarrow \varphi =$ *FF*
$\varphi \preceq$ *FVar x* $\longleftrightarrow \varphi =$ *FVar x*
⟨*proof*⟩

The variables inside the formula gives precisely the variables that are needed for the formula.

primrec *vars-of-prop*:: *'v propo* $\Rightarrow$ *'v set* **where**
vars-of-prop FT = {} |
vars-of-prop FF = {} |
vars-of-prop (*FVar x*) = $\{x\}$ |
vars-of-prop (*FNot* φ) = *vars-of-prop* φ |
vars-of-prop (*FAnd* $\varphi\ \psi$) = *vars-of-prop* $\varphi\ \cup$ *vars-of-prop* ψ |
vars-of-prop (*FOr* $\varphi\ \psi$) = *vars-of-prop* $\varphi\ \cup$ *vars-of-prop* ψ |
vars-of-prop (*FImp* $\varphi\ \psi$) = *vars-of-prop* $\varphi\ \cup$ *vars-of-prop* ψ |
vars-of-prop (*FEq* $\varphi\ \psi$) = *vars-of-prop* $\varphi\ \cup$ *vars-of-prop* ψ

lemma *vars-of-prop-incl-conn*:
fixes $\xi\ \xi'$ :: *'v propo list* **and** ψ :: *'v propo* **and** *c* :: *'v connective*
assumes *corr*: *wf-conn c l* **and** *incl*: $\psi \in$ *set l*

shows *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
⟨*proof*⟩

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:
$\varphi \preceq \psi \Longrightarrow$ *vars-of-prop* $\varphi \subseteq$ *vars-of-prop* ψ
⟨*proof*⟩

3.1.4 Positions

Instead of 1 or 2 we use L or R

datatype *sign* = L | R

We use *nil* instead of ε.

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

lemma *finite-pos*: *finite* (*pos* φ)
⟨*proof*⟩

lemma *finite-inj-comp-set*:
fixes *s* :: *'v set*
assumes *finite*: *finite s*
and *inj*: *inj f*
shows *card* ({*f p* |*p*. *p* ∈ *s*}) = *card s*
⟨*proof*⟩

lemma *cons-inject*:
inj (*op* # *s*)
⟨*proof*⟩

lemma *finite-insert-nil-cons*:
finite s $\Longrightarrow$ *card* (*insert* [] {*L* # *p* |*p*. *p* ∈ *s*}) = *1* + *card* {*L* # *p* |*p*. *p* ∈ *s*}
⟨*proof*⟩

lemma *cord-not*[*simp*]:
card (*pos* (*FNot* φ)) = *1* + *card* (*pos* φ)
⟨*proof*⟩

lemma *card-seperate*:
assumes *finite s1* **and** *finite s2*
shows *card* ({*L* # *p* |*p*. *p* ∈ *s1*} ∪ {*R* # *p* |*p*. *p* ∈ *s2*}) = *card* ({*L* # *p* |*p*. *p* ∈ *s1*})
+ *card*({*R* # *p* |*p*. *p* ∈ *s2*}) (**is** *card* (*?L*∪*?R*) = *card ?L* + *card ?R*)
⟨*proof*⟩

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

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

  ⟨*proof*⟩

value *pos* (*FImp* (*FAnd* (*FVar P*) (*FVar Q*)) (*FOr* (*FVar P*) (*FVar Q*)))

inductive *path-to* :: *sign list* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **where**
path-to-refl[*intro*]: *path-to* [] φ φ |
path-to-l: $c \in$*binary-connectives* $\vee$ c = *CNot* $\Longrightarrow$ *wf-conn* c (φ#l) $\Longrightarrow$ *path-to* p φ $\varphi'$$\Longrightarrow$
  path-to (L#p) (*conn* c (φ#l)) φ' |
path-to-r: $c \in$*binary-connectives* $\Longrightarrow$ *wf-conn* c (ψ#φ#[]) $\Longrightarrow$ *path-to* p φ φ' $\Longrightarrow$
  path-to (R#p) (*conn* c (ψ#φ#[])) φ'

There is a deep link between subformulas and pathes: a (correct) path leads to a subformula and a subformula is associated to a given path.

lemma *path-to-subformula*:
  path-to p φ φ' $\Longrightarrow$ $\varphi' \preceq \varphi$
  ⟨*proof*⟩

lemma *subformula-path-exists*:
  fixes φ φ':: *'v propo*
  shows $\varphi' \preceq \varphi \Longrightarrow \exists\, p.$ *path-to* p φ φ'
⟨*proof*⟩

fun *replace-at* :: *sign list* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* **where**
replace-at [] - ψ = ψ |
replace-at (L # l) (*FAnd* φ φ') ψ = *FAnd* (*replace-at* l φ ψ) φ'|
replace-at (R # l) (*FAnd* φ φ') ψ = *FAnd* φ (*replace-at* l φ' ψ) |
replace-at (L # l) (*FOr* φ φ') ψ = *FOr* (*replace-at* l φ ψ) φ' |
replace-at (R # l) (*FOr* φ φ') ψ = *FOr* φ (*replace-at* l φ' ψ) |
replace-at (L # l) (*FEq* φ φ') ψ = *FEq* (*replace-at* l φ ψ) φ'|
replace-at (R # l) (*FEq* φ φ') ψ = *FEq* φ (*replace-at* l φ' ψ) |
replace-at (L # l) (*FImp* φ φ') ψ = *FImp* (*replace-at* l φ ψ) φ'|
replace-at (R # l) (*FImp* φ φ') ψ = *FImp* φ (*replace-at* l φ' ψ) |
replace-at (L # l) (*FNot* φ) ψ = *FNot* (*replace-at* l φ ψ)

3.2 Semantics over the syntax

Given the syntax defined above, we define a semantics, by defining an evaluation function *eval*. This function is the bridge between the logic as we define it here and the built-in logic of Isabelle.

fun *eval* :: (*'v* $\Rightarrow$ *bool*) $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* (**infix** $\models$ *50*) **where**
$\mathcal{A} \models$ *FT* = *True* |
$\mathcal{A} \models$ *FF* = *False* |
$\mathcal{A} \models$ *FVar* v = ($\mathcal{A}$ v) |
$\mathcal{A} \models$ *FNot* φ = ($\neg$($\mathcal{A}\models$ φ)) |
$\mathcal{A} \models$ *FAnd* φ_1 φ_2 = ($\mathcal{A}\models\varphi_1$ $\wedge$ $\mathcal{A}\models\varphi_2$) |
$\mathcal{A} \models$ *FOr* φ_1 φ_2 = ($\mathcal{A}\models\varphi_1$ $\vee$ $\mathcal{A}\models\varphi_2$) |
$\mathcal{A} \models$ *FImp* φ_1 φ_2 = ($\mathcal{A}\models\varphi_1$ $\longrightarrow$ $\mathcal{A}\models\varphi_2$) |
$\mathcal{A} \models$ *FEq* φ_1 φ_2 = ($\mathcal{A}\models\varphi_1$ $\longleftrightarrow$ $\mathcal{A}$ $\models\varphi_2$)

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

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

theorem *deduction-theorem*:
$\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
⟨*proof*⟩

A shorter proof:

lemma $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
⟨*proof*⟩

definition *same-over-set*:: $('v \Rightarrow bool) \Rightarrow ('v \Rightarrow bool) \Rightarrow 'v\ set \Rightarrow bool$ **where**
$same\text{-}over\text{-}set\ A\ B\ S = (\forall c \in S.\ A\ c = B\ c)$

If two mapping A and B have the same value over the variables, then the same formula are satisfiable.

lemma *same-over-set-eval*:
assumes *same-over-set* A B (*vars-of-prop* φ)
shows $A \models \varphi \longleftrightarrow B \models \varphi$
⟨*proof*⟩

end
theory *Prop-Abstract-Transformation*
imports *Main Prop-Logic Wellfounded-More*

begin

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

3.3 Rewrite systems and properties

3.3.1 Lifting of rewrite rules

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

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

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

This lemma is only a health condition:

lemma *propo-rew-step-subformula-imp*:
shows $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$
⟨*proof*⟩

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

lemma *propo-rew-step-subformula-rec*:

fixes $\psi\ \psi'\ \varphi :: 'v\ propo$
shows $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists \varphi'.\ \psi' \preceq \varphi' \wedge \mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \varphi')$
$\langle proof \rangle$

lemma *propo-rew-step-subformula*:
$(\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi') \longleftrightarrow (\exists \varphi'.\ \mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \varphi')$
$\langle proof \rangle$

lemma *consistency-decompose-into-list*:
assumes *wf*: *wf-conn* $c\ l$ **and** *wf'*: *wf-conn* $c\ l'$
and *same*: $\forall n.\ A \models l\ !\ n \longleftrightarrow (A \models l'\ !\ n)$
shows $A \models \mathit{conn}\ c\ l \longleftrightarrow A \models \mathit{conn}\ c\ l'$
$\langle proof \rangle$

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

lemma *propo-rew-step-rewrite*:
fixes $\varphi\ \varphi' :: 'v\ propo$ **and** $r :: 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$
assumes *propo-rew-step* $r\ \varphi\ \varphi'$
shows $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge \mathit{path\text{-}to}\ p\ \varphi\ \psi \wedge \mathit{replace\text{-}at}\ p\ \varphi\ \psi' = \varphi'$
$\langle proof \rangle$

3.3.2 Consistency preservation

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

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

lemma *propo-rew-step-preservers-val-explicit*:
propo-rew-step $r\ \varphi\ \psi \Longrightarrow$ *preserves-un-sat* $r \Longrightarrow$ *propo-rew-step* $r\ \varphi\ \psi \Longrightarrow (\forall A.\ A \models \varphi \longleftrightarrow A \models \psi)$
$\langle proof \rangle$

lemma *propo-rew-step-preservers-val'*:
assumes *preserves-un-sat* r
shows *preserves-un-sat* (*propo-rew-step* r)
$\langle proof \rangle$

lemma *preserves-un-sat-OO*[*intro*]:
preserves-un-sat $f \Longrightarrow$ *preserves-un-sat* $g \Longrightarrow$ *preserves-un-sat* (f *OO* g)
$\langle proof \rangle$

lemma *star-consistency-preservation-explicit*:
assumes (*propo-rew-step* r)^** $\varphi\ \psi$ **and** *preserves-un-sat* r
shows $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
$\langle proof \rangle$

lemma *star-consistency-preservation*:
preserves-un-sat $r \Longrightarrow$ *preserves-un-sat* (*propo-rew-step* r)^**
$\langle proof \rangle$

3.3.3 Full Lifting

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

lemma *full-ropo-rew-step-preservers-val*[*simp*]:
preserves-un-sat $r \Longrightarrow$ *preserves-un-sat* (*full* (*propo-rew-step* r))
⟨*proof*⟩

lemma *full-propo-rew-step-subformula*:
full (*propo-rew-step* r) φ' $\varphi \Longrightarrow \neg(\exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi')$
⟨*proof*⟩

3.4 Transformation testing

3.4.1 Definition and first properties

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

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

lemma *test-symb-imp-all-subformula-st*[*simp*]:
test-symb $FT \Longrightarrow$ *all-subformula-st test-symb* FT
test-symb $FF \Longrightarrow$ *all-subformula-st test-symb* FF
test-symb ($FVar\ x$) $\Longrightarrow$ *all-subformula-st test-symb* ($FVar\ x$)
⟨*proof*⟩

lemma *all-subformula-st-test-symb-true-phi*:
all-subformula-st test-symb $\varphi \Longrightarrow$ *test-symb* φ
⟨*proof*⟩

lemma *all-subformula-st-decomp-imp*:
wf-conn $c\ l \Longrightarrow$ (*test-symb* (*conn* $c\ l$) $\wedge$ ($\forall\varphi\in$ *set* l. *all-subformula-st test-symb* φ))
$\Longrightarrow$ *all-subformula-st test-symb* (*conn* $c\ l$)
⟨*proof*⟩

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

lemma *all-subformula-st-decomp-rec*:
all-subformula-st test-symb (*conn* $c\ l$) $\Longrightarrow$ *wf-conn* $c\ l$
$\Longrightarrow$ (*test-symb* (*conn* $c\ l$) $\wedge$ ($\forall\varphi\in$ *set* l. *all-subformula-st test-symb* φ))
⟨*proof*⟩

lemma *all-subformula-st-decomp*:
fixes c :: $'v$ *connective* **and** l :: $'v$ *propo list*
assumes *wf-conn* $c\ l$
shows *all-subformula-st test-symb* (*conn* $c\ l$)
$\longleftrightarrow$ (*test-symb* (*conn* $c\ l$) $\wedge$ ($\forall\varphi\in$ *set* l. *all-subformula-st test-symb* φ))
⟨*proof*⟩

lemma *helper-fact*: $c \in$ *binary-connectives* $\longleftrightarrow (c = COr \vee c = CAnd \vee c = CEq \vee c = CImp)$
⟨*proof*⟩
lemma *all-subformula-st-decomp-explicit*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*
shows *all-subformula-st test-symb* $(FAnd\ \varphi\ \psi)$
$\longleftrightarrow$ (*test-symb* $(FAnd\ \varphi\ \psi) \wedge$ *all-subformula-st test-symb* $\varphi \wedge$ *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb* $(FOr\ \varphi\ \psi)$
$\longleftrightarrow$ (*test-symb* $(FOr\ \varphi\ \psi) \wedge$ *all-subformula-st test-symb* $\varphi \wedge$ *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb* $(FNot\ \varphi)$
$\longleftrightarrow$ (*test-symb* $(FNot\ \varphi) \wedge$ *all-subformula-st test-symb* φ)
and *all-subformula-st test-symb* $(FEq\ \varphi\ \psi)$
$\longleftrightarrow$ (*test-symb* $(FEq\ \varphi\ \psi) \wedge$ *all-subformula-st test-symb* $\varphi \wedge$ *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb* $(FImp\ \varphi\ \psi)$
$\longleftrightarrow$ (*test-symb* $(FImp\ \varphi\ \psi) \wedge$ *all-subformula-st test-symb* $\varphi \wedge$ *all-subformula-st test-symb* ψ)
⟨*proof*⟩

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

lemma *subformula-all-subformula-st*:
$\psi \preceq \varphi \Longrightarrow$ *all-subformula-st test-symb* $\varphi \Longrightarrow$ *all-subformula-st test-symb* ψ
⟨*proof*⟩

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

lemma *no-test-symb-step-exists*:
fixes r:: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **and** *test-symb*:: *'v propo* $\Rightarrow$ *bool* **and** x :: *'v*
and φ :: *'v propo*
assumes
test-symb-false-nullary: $\forall x.$ *test-symb* $FF \wedge$ *test-symb* $FT \wedge$ *test-symb* $(FVar\ x)$ **and**
$\forall \varphi'.\ \varphi' \preceq \varphi \longrightarrow (\neg$*test-symb* $\varphi') \longrightarrow (\exists\ \psi.\ r\ \varphi'\ \psi)$ **and**
$\neg$ *all-subformula-st test-symb* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
⟨*proof*⟩

3.4.2 Invariant conservation

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

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

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

Invariant while lifting of the rewriting relation

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

lemma *propo-rew-step-inv-stay'*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and φ ψ Φ:: $'v$ *propo*
assumes H: $\forall \varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* φ'
$\longrightarrow$ *all-subformula-st test-symb* ψ
and H': $\forall$ (c:: $'v$ *connective*) $\xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi'$
$\longrightarrow$ *wf-conn* c $(\xi$ @ φ # $\xi')$ $\longrightarrow$ *test-symb* (*conn* c $(\xi$ @ φ # $\xi'))$ $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn* c $(\xi$ @ φ' # $\xi'))$ **and**
propo-rew-step $r\ \varphi\ \psi$ **and**
$\varphi \preceq \Phi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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

lemma *propo-rew-step-inv-stay*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and φ ψ :: $'v$ *propo*
assumes
H: $\forall \varphi'\ \psi.\ r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi' \longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (c:: $'v$ *connective*) $\xi\ \varphi\ \xi'\ \varphi'$. *wf-conn* c $(\xi$ @ φ # $\xi')$ $\longrightarrow$ *test-symb* (*conn* c $(\xi$ @ φ # $\xi'))$
$\longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn* c $(\xi$ @ φ' # $\xi'))$ **and**
propo-rew-step $r\ \varphi\ \psi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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

Invariant after all rewriting

lemma *full-propo-rew-step-inv-stay-with-inc*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and φ ψ :: $'v$ *propo*
assumes
H: $\forall\ \varphi\ \psi$. *propo-rew-step* $r\ \varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (c:: $'v$ *connective*) $\xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi'$
$\longrightarrow$ *wf-conn* c $(\xi$ @ φ # $\xi')$ $\longrightarrow$ *test-symb* (*conn* c $(\xi$ @ φ # $\xi'))$ $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn* c $(\xi$ @ φ' # $\xi'))$ **and**
$\varphi \preceq \Phi$ **and**
full: *full* (*propo-rew-step* r) $\varphi\ \psi$ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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

$\longrightarrow$ *test-symb* (*conn c* (ξ @ φ # ξ')) $\longrightarrow$ *test-symb* φ' $\longrightarrow$ *test-symb* (*conn c* (ξ @ φ' # ξ')) **and**
full: *full* (*propo-rew-step r*) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

lemma *full-propo-rew-step-inv-stay*:
fixes *r*:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. *r* φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) ξ φ ξ' φ'. *wf-conn c* (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn c* (ξ @ φ # ξ'))
$\longrightarrow$ *test-symb* φ' $\longrightarrow$ *test-symb* (*conn c* (ξ @ φ' # ξ')) **and**
full: *full* (*propo-rew-step r*) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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

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

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

3.5 Rewrite Rules

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

3.5.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

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

lemma *elim-equiv-transformation-consistent*:
A ⊨ *FEq* φ ψ $\longleftrightarrow$ *A* ⊨ *FAnd* (*FImp* φ ψ) (*FImp* ψ φ)

⟨*proof*⟩

lemma *elim-equiv-explicit*: *elim-equiv* φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*proof*⟩

lemma *elim-equiv-consistent*: *preserves-un-sat elim-equiv*
⟨*proof*⟩

lemma *elimEquv-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step elim-equiv*))
⟨*proof*⟩

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

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

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

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

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

lemma *no-equiv-eq*[*simp*]:
fixes φ ψ :: *'v propo*
shows
$\neg$*no-equiv* (*FEq* φ ψ)
no-equiv FT
no-equiv FF
⟨*proof*⟩

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

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

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

lemma *no-equiv-elim-equiv-step*:
fixes φ :: *'v propo*
assumes *no-equiv*: $\neg$ *no-equiv* φ
shows $\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elim-equiv* ψ ψ'
⟨*proof*⟩

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

lemma *no-equiv-full-propo-rew-step-elim-equiv*:
  full (*propo-rew-step elim-equiv*) φ ψ $\Longrightarrow$ *no-equiv* ψ
  ⟨*proof*⟩

3.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

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

lemma *elim-imp-transformation-consistent*:
  $A \models$ *FImp* φ ψ $\longleftrightarrow$ $A \models$ *FOr* (*FNot* φ) ψ
  ⟨*proof*⟩

lemma *elim-imp-explicit*: *elim-imp* φ ψ $\Longrightarrow$ $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
  ⟨*proof*⟩

lemma *elim-imp-consistent*: *preserves-un-sat elim-imp*
  ⟨*proof*⟩

lemma *elim-imp-lifted-consistant*:
  preserves-un-sat (*full* (*propo-rew-step elim-imp*))
  ⟨*proof*⟩

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

lemma *no-imp-symb-conn-characterization*:
  wf-conn c l $\Longrightarrow$ *no-imp-symb* (*conn c l*) $\longleftrightarrow$ $c \neq$ *CImp*
  ⟨*proof*⟩

definition *no-imp* **where** *no-imp* $\equiv$ *all-subformula-st no-imp-symb*
declare *no-imp-def*[*simp*]

lemma *no-imp-Imp*[*simp*]:
  $\neg$*no-imp* (*FImp* φ ψ)
  no-imp FT
  no-imp FF
  ⟨*proof*⟩

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

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
  elim-imp φ ψ $\Longrightarrow$ *no-equiv* φ $\Longrightarrow$ *no-equiv* ψ

$\langle proof \rangle$

lemma *elim-imp-inv*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes *full* (*propo-rew-step elim-imp*) $\varphi\ \psi$ **and** *no-equiv* φ
shows *no-equiv* ψ
$\langle proof \rangle$

lemma *no-no-imp-elim-imp-step-exists*:
fixes φ :: *'v propo*
assumes *no-equiv*: $\neg$ *no-imp* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elim-imp* $\psi\ \psi'$
$\langle proof \rangle$

lemma *no-imp-full-propo-rew-step-elim-imp*: *full* (*propo-rew-step elim-imp*) $\varphi\ \psi \Longrightarrow$ *no-imp* ψ
$\langle proof \rangle$

3.5.3 Eliminate all the True and False in the formula

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

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

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

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

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

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

lemma *elimTB-consistent*: *preserves-un-sat elimTB*
$\langle proof \rangle$

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

lemma *wf-conn-no-T-F-symb-iff*[*simp*]:
wf-conn $c\ \psi s \Longrightarrow$
no-T-F-symb (*conn* $c\ \psi s$) $\longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in set\ \psi s.\ \psi \neq FF \wedge \psi \neq FT))$
$\langle proof \rangle$

lemma *wf-conn-no-T-F-symb-iff-explicit*[*simp*]:
no-T-F-symb (*FAnd* $\varphi\ \psi$) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FOr* $\varphi\ \psi$) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FEq* $\varphi\ \psi$) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$

$no\text{-}T\text{-}F\text{-}symb\ (FImp\ \varphi\ \psi) \longleftrightarrow (\forall\chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
⟨*proof*⟩

lemma *no-T-F-symb-false*[*simp*]:
fixes $c :: 'v\ connective$
shows
$\neg no\text{-}T\text{-}F\text{-}symb\ (FT :: 'v\ propo)$
$\neg no\text{-}T\text{-}F\text{-}symb\ (FF :: 'v\ propo)$
⟨*proof*⟩

lemma *no-T-F-symb-bool*[*simp*]:
fixes $x :: 'v$
shows $no\text{-}T\text{-}F\text{-}symb\ (FVar\ x)$
⟨*proof*⟩

lemma *no-T-F-symb-fnot-imp*:
$\neg no\text{-}T\text{-}F\text{-}symb\ (FNot\ \varphi) \Longrightarrow \varphi = FT \vee \varphi = FF$
⟨*proof*⟩

lemma *no-T-F-symb-fnot*[*simp*]:
$no\text{-}T\text{-}F\text{-}symb\ (FNot\ \varphi) \longleftrightarrow \neg(\varphi = FT \vee \varphi = FF)$
⟨*proof*⟩

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

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

lemma *no-T-F-symb-except-toplevel-bool*:
fixes $x :: 'v$
shows $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\ (FVar\ x)$
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-not-decom*:
$\varphi \neq FT \Longrightarrow \varphi \neq FF \Longrightarrow no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\ (FNot\ \varphi)$
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes $\varphi\ \psi :: 'v\ propo$
assumes $\varphi \neq FT$ **and** $\varphi \neq FF$ **and** $\psi \neq FT$ **and** $\psi \neq FF$
and c: $c \in binary\text{-}connectives$
shows $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\ (conn\ c\ [\varphi, \psi])$
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:
fixes $l :: 'v\ propo\ list$ **and** $c :: 'v\ connective$
assumes *corr*: *wf-conn c l*
and $FT \in set\ l \vee FF \in set\ l$
shows $\neg no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\ (conn\ c\ l)$
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-example*[*simp*]:
fixes φ ψ :: *'v propo*
assumes $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FAnd* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FOr* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FImp* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FEq* φ ψ)
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-not*[*simp*]:
fixes φ ψ :: *'v propo*
assumes $\varphi = FT \lor \varphi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FNot* φ)
⟨*proof*⟩

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

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

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

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

lemma *no-T-F-except-top-level-false*:
fixes l :: *'v propo list* **and** c :: *'v connective*
assumes *wf-conn c l*
and $FT \in set\ l \lor FF \in set\ l$
shows $\neg$*no-T-F-except-top-level* (*conn c l*)
⟨*proof*⟩

lemma *no-T-F-except-top-level-false-example*[*simp*]:
fixes φ ψ :: *'v propo*
assumes $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
$\neg$*no-T-F-except-top-level* (*FAnd* φ ψ)
$\neg$*no-T-F-except-top-level* (*FOr* φ ψ)
$\neg$*no-T-F-except-top-level* (*FEq* φ ψ)
$\neg$*no-T-F-except-top-level* (*FImp* φ ψ)
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-no-T-F-symb*:
no-T-F-symb-except-toplevel $\varphi \Longrightarrow \varphi \neq FF \Longrightarrow \varphi \neq FT \Longrightarrow$ *no-T-F-symb* φ
⟨*proof*⟩

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

lemma *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*:
no-T-F-except-top-level $\varphi \Longrightarrow \varphi \neq FF \Longrightarrow \varphi \neq FT \Longrightarrow$ *no-T-F* φ
⟨*proof*⟩

lemma *no-T-F-no-T-F-except-top-level*:
no-T-F $\varphi \Longrightarrow$ *no-T-F-except-top-level* φ

$\langle proof \rangle$

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level FF no-T-F-except-top-level FT*
$\langle proof \rangle$

lemma *no-T-F-no-T-F-except-top-level'*[*simp*]:
no-T-F-except-top-level $\varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee$ *no-T-F* $\varphi)$
$\langle proof \rangle$

lemma *no-T-F-bin-decomp*[*simp*]:
assumes *c*: $c \in$ *binary-connectives*
shows *no-T-F* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
$\langle proof \rangle$

lemma *no-T-F-bin-decomp-expanded*[*simp*]:
assumes *c*: $c = CAnd \vee c = COr \vee c = CEq \vee c = CImp$
shows *no-T-F* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
$\langle proof \rangle$

lemma *no-T-F-comp-expanded-explicit*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*
shows
no-T-F (*FAnd* $\varphi\ \psi$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
no-T-F (*FOr* $\varphi\ \psi$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
no-T-F (*FEq* $\varphi\ \psi$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
no-T-F (*FImp* $\varphi\ \psi$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
$\langle proof \rangle$

lemma *no-T-F-comp-not*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*
shows *no-T-F* (*FNot* φ) $\longleftrightarrow$ *no-T-F* φ
$\langle proof \rangle$

lemma *no-T-F-decomp*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes φ: *no-T-F* (*FAnd* $\varphi\ \psi$) $\vee$ *no-T-F* (*FOr* $\varphi\ \psi$) $\vee$ *no-T-F* (*FEq* $\varphi\ \psi$) $\vee$ *no-T-F* (*FImp* $\varphi\ \psi$)
shows *no-T-F* ψ **and** *no-T-F* φ
$\langle proof \rangle$

lemma *no-T-F-decomp-not*:
fixes φ :: *'v propo*
assumes φ: *no-T-F* (*FNot* φ)
shows *no-T-F* φ
$\langle proof \rangle$

lemma *no-T-F-symb-except-toplevel-step-exists*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes *no-equiv* φ **and** *no-imp* φ
shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'.$ *elimTB* $\psi\ \psi'$
$\langle proof \rangle$

lemma *no-T-F-except-top-level-rew*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$ *no-T-F-except-top-level* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elimTB* $\psi\ \psi'$
$\langle proof \rangle$

lemma *elimTB-inv*:
fixes φ ψ :: $'v$ *propo*
assumes *full* (*propo-rew-step elimTB*) φ ψ
and *no-equiv* φ **and** *no-imp* φ
shows *no-equiv* ψ **and** *no-imp* ψ
$\langle proof \rangle$

lemma *elimTB-full-propo-rew-step*:
fixes φ ψ :: $'v$ *propo*
assumes *no-equiv* φ **and** *no-imp* φ **and** *full* (*propo-rew-step elimTB*) φ ψ
shows *no-T-F-except-top-level* ψ
$\langle proof \rangle$

3.5.4 PushNeg

Push the negation inside the formula, until the litteral.

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

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

lemma *pushNeg-explicit*: $pushNeg\ \varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
$\langle proof \rangle$

lemma *pushNeg-consistent*: *preserves-un-sat pushNeg*
$\langle proof \rangle$

lemma *pushNeg-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step pushNeg*))
$\langle proof \rangle$

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

lemma *simple-decomp*:
$simple\ \varphi \longleftrightarrow (\varphi = FT \lor \varphi = FF \lor (\exists x.\ \varphi = FVar\ x))$
$\langle proof \rangle$

lemma *subformula-conn-decomp-simple*:
fixes φ ψ :: $'v$ *propo*
assumes *s*: *simple* ψ
shows $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \lor \varphi = \psi)$

$\langle proof \rangle$

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

fun *simple-not-symb* **where**
simple-not-symb $(FNot\ \varphi) = (simple\ \varphi)$ |
simple-not-symb - = *True*

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

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

lemma *simple-not-step-exists*:
  fixes $\varphi\ \psi$ :: *'v propo*
  assumes *no-equiv* φ **and** *no-imp* φ
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *simple-not-symb* $\psi \Longrightarrow \exists \psi'.$ *pushNeg* $\psi\ \psi'$
  $\langle proof \rangle$

lemma *simple-not-rew*:
  fixes φ :: *'v propo*
  assumes *noTB*: $\neg$ *simple-not* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
  shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *pushNeg* $\psi\ \psi'$
$\langle proof \rangle$

lemma *no-T-F-except-top-level-pushNeg1*:
  no-T-F-except-top-level $(FNot\ (FAnd\ \varphi\ \psi)) \Longrightarrow$ *no-T-F-except-top-level* $(FOr\ (FNot\ \varphi)\ (FNot\ \psi))$
  $\langle proof \rangle$

lemma *no-T-F-except-top-level-pushNeg2*:
  no-T-F-except-top-level $(FNot\ (FOr\ \varphi\ \psi)) \Longrightarrow$ *no-T-F-except-top-level* $(FAnd\ (FNot\ \varphi)\ (FNot\ \psi))$
  $\langle proof \rangle$

lemma *no-T-F-symb-pushNeg*:
  no-T-F-symb $(FOr\ (FNot\ \varphi')\ (FNot\ \psi'))$
  no-T-F-symb $(FAnd\ (FNot\ \varphi')\ (FNot\ \psi'))$
  no-T-F-symb $(FNot\ (FNot\ \varphi'))$
  $\langle proof \rangle$

lemma *propo-rew-step-pushNeg-no-T-F-symb*:
  propo-rew-step pushNeg $\varphi\ \psi \Longrightarrow$ *no-T-F-except-top-level* $\varphi \Longrightarrow$ *no-T-F-symb* $\varphi \Longrightarrow$ *no-T-F-symb* ψ
  $\langle proof \rangle$

lemma *propo-rew-step-pushNeg-no-T-F*:
  propo-rew-step pushNeg $\varphi\ \psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ

⟨*proof*⟩

lemma *pushNeg-inv*:
  fixes φ ψ :: $'v$ *propo*
  assumes *full* (*propo-rew-step pushNeg*) φ ψ
  and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ
  shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ
⟨*proof*⟩

lemma *pushNeg-full-propo-rew-step*:
  fixes φ ψ :: $'v$ *propo*
  assumes
    no-equiv φ **and**
    no-imp φ **and**
    full (*propo-rew-step pushNeg*) φ ψ **and**
    no-T-F-except-top-level φ
  shows *simple-not* ψ
  ⟨*proof*⟩

3.5.5 Push inside

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

lemma *push-conn-inside-explicit*: *push-conn-inside* c c' φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
  ⟨*proof*⟩

lemma *push-conn-inside-consistent*: *preserves-un-sat* (*push-conn-inside* c c')
  ⟨*proof*⟩

lemma *propo-rew-step-push-conn-inside*[*simp*]:
 $\neg$*propo-rew-step* (*push-conn-inside* c c') *FT* ψ $\neg$*propo-rew-step* (*push-conn-inside* c c') *FF* ψ
 ⟨*proof*⟩

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

abbreviation *c-in-c'-symb* c c' φ $\equiv$ $\neg$*not-c-in-c'-symb* c c' φ

lemma *c-in-c'-symb-simp*:
  not-c-in-c'-symb c c' $\xi \Longrightarrow \xi = FF \lor \xi = FT \lor \xi = FVar\ x \lor \xi = FNot\ FF \lor \xi = FNot\ FT$
    $\lor\ \xi = FNot\ (FVar\ x) \Longrightarrow False$

$\langle proof \rangle$

lemma *c-in-c′-symb-simp′*[*simp*]:
$\neg$*not-c-in-c′-symb c c′ FF*
$\neg$*not-c-in-c′-symb c c′ FT*
$\neg$*not-c-in-c′-symb c c′* (*FVar x*)
$\neg$*not-c-in-c′-symb c c′* (*FNot FF*)
$\neg$*not-c-in-c′-symb c c′* (*FNot FT*)
$\neg$*not-c-in-c′-symb c c′* (*FNot* (*FVar x*))
$\langle proof \rangle$

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

lemma *c-in-c′-only-simp*[*simp*]:
c-in-c′-only c c′ FF
c-in-c′-only c c′ FT
c-in-c′-only c c′ (*FVar x*)
c-in-c′-only c c′ (*FNot FF*)
c-in-c′-only c c′ (*FNot FT*)
c-in-c′-only c c′ (*FNot* (*FVar x*))
$\langle proof \rangle$

lemma *not-c-in-c′-symb-commute*:
not-c-in-c′-symb c c′ $\xi \Longrightarrow$ *wf-conn c* $[\varphi, \psi] \Longrightarrow \xi =$ *conn c* $[\varphi, \psi]$
$\Longrightarrow$ *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
$\langle proof \rangle$

lemma *not-c-in-c′-symb-commute′*:
wf-conn c $[\varphi, \psi] \Longrightarrow$ *c-in-c′-symb c c′* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ *c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
$\langle proof \rangle$

lemma *not-c-in-c′-comm*:
assumes *wf*: *wf-conn c* $[\varphi, \psi]$
shows *c-in-c′-only c c′* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ *c-in-c′-only c c′* (*conn c* $[\psi, \varphi]$) (**is** *?A* $\longleftrightarrow$ *?B*)
$\langle proof \rangle$

lemma *not-c-in-c′-simp*[*simp*]:
fixes $\varphi 1$ $\varphi 2$ ψ :: *′v propo* **and** *x* :: *′v*
shows
c-in-c′-symb c c′ FT
c-in-c′-symb c c′ FF
c-in-c′-symb c c′ (*FVar x*)
wf-conn c [*conn c′* $[\varphi 1, \varphi 2], \psi] \Longrightarrow$ *wf-conn c′* $[\varphi 1, \varphi 2]$
$\Longrightarrow \neg$ *c-in-c′-only c c′* (*conn c* [*conn c′* $[\varphi 1, \varphi 2], \psi]$)
$\langle proof \rangle$

lemma *c-in-c′-symb-not*[*simp*]:
fixes *c c′* :: *′v connective* **and** ψ :: *′v propo*
shows *c-in-c′-symb c c′* (*FNot* ψ)
$\langle proof \rangle$

lemma *c-in-c′-symb-step-exists*:
fixes φ :: *′v propo*
assumes *c*: *c* = *CAnd* $\vee$ *c* = *COr* **and** *c′*: *c′* = *CAnd* $\vee$ *c′* = *COr*

shows $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c′-symb c c′* $\psi \Longrightarrow \exists \psi'$. *push-conn-inside c c′* ψ ψ'
⟨*proof*⟩

lemma *c-in-c′-symb-rew*:
fixes φ :: *′v propo*
assumes *noTB*: ¬*c-in-c′-only c c′* φ
and *c*: $c = CAnd \vee c = COr$ **and** *c′*: $c' = CAnd \vee c' = COr$
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *push-conn-inside c c′* ψ ψ'
⟨*proof*⟩

lemma *push-conn-insidec-in-c′-symb-no-T-F*:
fixes φ ψ :: *′v propo*
shows *propo-rew-step* (*push-conn-inside c c′*) φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

lemma *simple-propo-rew-step-push-conn-inside-inv*:
propo-rew-step (*push-conn-inside c c′*) φ $\psi \Longrightarrow$ *simple* $\varphi \Longrightarrow$ *simple* ψ
⟨*proof*⟩

lemma *simple-propo-rew-step-inv-push-conn-inside-simple-not*:
fixes *c c′* :: *′v connective* **and** φ ψ :: *′v propo*
shows *propo-rew-step* (*push-conn-inside c c′*) φ $\psi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *simple-not* ψ
⟨*proof*⟩

lemma *propo-rew-step-push-conn-inside-simple-not*:
fixes φ φ' :: *′v propo* **and** ξ ξ' :: *′v propo list* **and** *c* :: *′v connective*
assumes
propo-rew-step (*push-conn-inside c c′*) φ φ' **and**
wf-conn c (ξ @ φ # ξ') **and**
simple-not-symb (*conn c* (ξ @ φ # ξ')) **and**
simple-not-symb φ'
shows *simple-not-symb* (*conn c* (ξ @ φ' # ξ'))
⟨*proof*⟩

lemma *push-conn-inside-not-true-false*:
push-conn-inside c c′ φ $\psi \Longrightarrow \psi \neq FT \wedge \psi \neq FF$
⟨*proof*⟩

lemma *push-conn-inside-inv*:
fixes φ ψ :: *′v propo*
assumes *full* (*propo-rew-step* (*push-conn-inside c c′*)) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
⟨*proof*⟩

lemma *push-conn-inside-full-propo-rew-step*:
fixes φ ψ :: *′v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step* (*push-conn-inside c c′*)) φ ψ **and**
no-T-F-except-top-level φ **and**

simple-not φ **and**
$c = CAnd \lor c = COr$ **and**
$c' = CAnd \lor c' = COr$
shows *c-in-c'-only* $c\ c'\ \psi$
⟨*proof*⟩

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

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

lemma *only-c-inside-symb-simp*[*simp*]:
only-c-inside-symb c *FF* *only-c-inside-symb* c *FT* *only-c-inside-symb* c (*FVar* x) ⟨*proof*⟩

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

lemma *only-c-inside-symb-decomp*:
only-c-inside-symb $c\ \psi \longleftrightarrow$ (*simple* ψ
$\lor$ ($\exists\ \varphi'$. ψ = *FNot* $\varphi' \land$ *simple* φ')
$\lor$ ($\exists\, l$. ψ = *conn* $c\ l \land$ *wf-conn* $c\ l$))
⟨*proof*⟩

lemma *only-c-inside-symb-decomp-not*[*simp*]:
fixes c :: $'v$ *connective*
assumes c: $c \neq CNot$
shows *only-c-inside-symb* c (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
⟨*proof*⟩

lemma *only-c-inside-decomp-not*[*simp*]:
assumes c: $c \neq CNot$
shows *only-c-inside* c (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
⟨*proof*⟩

lemma *only-c-inside-decomp*:
only-c-inside $c\ \varphi \longleftrightarrow$
($\forall\,\psi$. $\psi \preceq \varphi \longrightarrow$ (*simple* $\psi \lor$ ($\exists\ \varphi'$. ψ = *FNot* $\varphi' \land$ *simple* φ')
$\lor$ ($\exists\, l$. ψ = *conn* $c\ l \land$ *wf-conn* $c\ l$)))
⟨*proof*⟩

lemma *only-c-inside-c-c'-false*:
fixes $c\ c'$ :: $'v$ *connective* **and** l :: $'v$ *propo list* **and** φ :: $'v$ *propo*
assumes *cc'*: $c \neq c'$ **and** c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
and *only*: *only-c-inside* $c\ \varphi$ **and** *incl*: *conn* $c'\ l \preceq \varphi$ **and** *wf*: *wf-conn* $c'\ l$
shows *False*
⟨*proof*⟩

lemma *only-c-inside-implies-c-in-c'-symb*:
assumes δ: $c \neq c'$ **and** c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
shows *only-c-inside* $c\ \varphi \Longrightarrow$ *c-in-c'-symb* $c\ c'\ \varphi$
⟨*proof*⟩

lemma *c-in-c′-symb-decomp-level1*:
  fixes *l* :: *′v propo list* **and** *c c′ ca* :: *′v connective*
  shows *wf-conn ca l* $\Longrightarrow$ *ca* $\neq$ *c* $\Longrightarrow$ *c-in-c′-symb c c′* (*conn ca l*)
⟨*proof*⟩

lemma *only-c-inside-implies-c-in-c′-only*:
  assumes δ: $c \neq c'$ **and** *c*: $c = CAnd \lor c = COr$ **and** *c′*: $c' = CAnd \lor c' = COr$
  shows *only-c-inside c* φ $\Longrightarrow$ *c-in-c′-only c c′* φ
  ⟨*proof*⟩

lemma *c-in-c′-symb-c-implies-only-c-inside*:
  assumes δ: $c = CAnd \lor c = COr$ $c' = CAnd \lor c' = COr$ $c \neq c'$ **and** *wf*: *wf-conn c* [φ, ψ]
  and *inv*: *no-equiv* (*conn c l*) *no-imp* (*conn c l*) *simple-not* (*conn c l*)
  shows *wf-conn c l* $\Longrightarrow$ *c-in-c′-only c c′* (*conn c l*) $\Longrightarrow$ ($\forall\,\psi\in$ *set l. only-c-inside c* ψ)
⟨*proof*⟩

Push Conjunction

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

lemma *pushConj-consistent*: *preserves-un-sat pushConj*
  ⟨*proof*⟩

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

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

lemma *pushConj-inv*:
  fixes φ ψ :: *′v propo*
  assumes *full* (*propo-rew-step pushConj*) φ ψ
  and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
  shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
  ⟨*proof*⟩

lemma *pushConj-full-propo-rew-step*:
  fixes φ ψ :: *′v propo*
  assumes
    no-equiv φ **and**
    no-imp φ **and**
    full (*propo-rew-step pushConj*) φ ψ **and**
    no-T-F-except-top-level φ **and**
    simple-not φ
  shows *and-in-or-only* ψ
  ⟨*proof*⟩

Push Disjunction

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

lemma *pushDisj-consistent*: *preserves-un-sat pushDisj*
  ⟨*proof*⟩

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

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

lemma *not-or-in-and-only-or-and*[*simp*]:
 ~*or-in-and-only* (*FOr* (*FAnd* ψ*1* ψ*2*) φ')
 ⟨*proof*⟩

lemma *pushDisj-inv*:
 fixes φ ψ :: ′*v propo*
 assumes *full* (*propo-rew-step pushDisj*) φ ψ
 and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
 shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
 ⟨*proof*⟩

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

3.6 The full transformations

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

Definition

The normal is a super group of groups

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

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

lemma *only-c-inside-symb-c-eq-c′*:
 only-c-inside-symb c (*conn c′* [φ*1*, φ*2*]) ⟹ *c′* = *CAnd* ∨ *c′* = *COr* ⟹ *wf-conn c′* [φ*1*, φ*2*]
  ⟹ *c′* = *c*
 ⟨*proof*⟩

lemma *only-c-inside-c-eq-c'*:
only-c-inside c $(conn\ c'\ [\varphi 1,\ \varphi 2]) \Longrightarrow\ c' = CAnd \lor c' = COr \Longrightarrow wf\text{-}conn\ c'\ [\varphi 1,\ \varphi 2] \Longrightarrow c = c'$
⟨*proof*⟩

lemma *only-c-inside-imp-grouped-by*:
assumes c: $c \neq CNot$ **and** c': $c' = CAnd \lor c' = COr$
shows *only-c-inside* $c\ \varphi \Longrightarrow$ *grouped-by* $c\ \varphi$ (**is** $?O\ \varphi \Longrightarrow ?G\ \varphi$)
⟨*proof*⟩

lemma *grouped-by-false*:
grouped-by $c\ (conn\ c'\ [\varphi,\ \psi]) \Longrightarrow c \neq c' \Longrightarrow wf\text{-}conn\ c'\ [\varphi,\ \psi] \Longrightarrow False$
⟨*proof*⟩

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

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

lemma *simple-cnf*[*simp*]:
super-grouped-by $c\ c'$ *FT*
super-grouped-by $c\ c'$ *FF*
super-grouped-by $c\ c'$ (*FVar* x)
super-grouped-by $c\ c'$ (*FNot FT*)
super-grouped-by $c\ c'$ (*FNot FF*)
super-grouped-by $c\ c'$ (*FNot* (*FVar* x))
⟨*proof*⟩

lemma *c-in-c'-only-super-grouped-by*:
assumes c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$ **and** cc': $c \neq c'$
shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *c-in-c'-only* $c\ c'\ \varphi$
$\Longrightarrow$ *super-grouped-by* $c\ c'\ \varphi$
(**is** $?NE\ \varphi \Longrightarrow ?NI\ \varphi \Longrightarrow ?SN\ \varphi \Longrightarrow ?C\ \varphi \Longrightarrow ?S\ \varphi$)
⟨*proof*⟩

3.6.2 Conjunctive Normal Form

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

lemma *or-in-and-only-conjunction-in-disj*:
shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *or-in-and-only* $\varphi \Longrightarrow$ *is-conj-with-TF* φ
⟨*proof*⟩

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

Full CNF transformation

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

definition *cnf-rew* **where** *cnf-rew* =
(*full* (*propo-rew-step elim-equiv*)) *OO*

(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step elimTB*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushDisj*))

lemma *cnf-rew-consistent*: *preserves-un-sat cnf-rew*
⟨*proof*⟩

lemma *cnf-rew-is-cnf*: *cnf-rew* φ φ' $\Longrightarrow$ *is-cnf* φ'
⟨*proof*⟩

3.6.3 Disjunctive Normal Form

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

lemma *and-in-or-only-conjunction-in-disj*:
shows *no-equiv* φ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *simple-not* φ $\Longrightarrow$ *and-in-or-only* φ $\Longrightarrow$ *is-disj-with-TF* φ
⟨*proof*⟩

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

Full DNF transform

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

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

lemma *dnf-rew-consistent*: *preserves-un-sat dnf-rew*
⟨*proof*⟩

theorem *dnf-transformation-correction*:
dnf-rew φ φ' $\Longrightarrow$ *is-dnf* φ'
⟨*proof*⟩

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

3.7.1 Transformation

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

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

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

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

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

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

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

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

The transformation is still consistent.

lemma *elimTBFull-consistent*: *preserves-un-sat elimTBFull*
⟨*proof*⟩

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

lemma *no-T-F-symb-except-toplevel-step-exists'*:
  fixes φ :: *'v propo*
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'.$ *elimTBFull* ψ ψ'
⟨*proof*⟩

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

lemma *no-T-F-except-top-level-rew'*:
  fixes φ :: *'v propo*
  assumes *noTB*: ¬ *no-T-F-except-top-level* φ
  shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elimTBFull* ψ ψ'
⟨*proof*⟩

lemma *elimTBFull-full-propo-rew-step*:
  fixes φ ψ :: *'v propo*
  assumes *full* (*propo-rew-step elimTBFull*) φ ψ
  shows *no-T-F-except-top-level* ψ
  ⟨*proof*⟩

3.7.2 More invariants

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

lemma *propo-rew-step-ElimEquiv-no-T-F*: *propo-rew-step elim-equiv* φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

lemma *elim-equiv-inv'*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-equiv*) φ ψ **and** *no-T-F-except-top-level* φ
shows *no-T-F-except-top-level* ψ
⟨*proof*⟩

lemma *propo-rew-step-ElimImp-no-T-F*: *propo-rew-step elim-imp* φ ψ $\Longrightarrow$ *no-T-F* φ $\Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

lemma *elim-imp-inv'*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-imp*) φ ψ **and** *no-T-F-except-top-level* φ
shows*no-T-F-except-top-level* ψ
⟨*proof*⟩

3.7.3 The new CNF and DNF transformation

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

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

lemma *dnf-rew'-consistent*: *preserves-un-sat dnf-rew'*
⟨*proof*⟩

theorem *cnf-transformation-correction*:
dnf-rew' φ φ' $\Longrightarrow$ *is-dnf* φ'
⟨*proof*⟩

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

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

lemma *cnf-rew'-consistent*: *preserves-un-sat cnf-rew'*
⟨*proof*⟩

theorem *cnf'-transformation-correction*:
cnf-rew' φ φ' $\Longrightarrow$ *is-cnf* φ'
⟨*proof*⟩

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

3.8 Link with Multiset Version

3.8.1 Transformation to Multiset

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

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

3.8.2 Equisatisfiability of the two Version

lemma *is-conj-with-TF-FNot*:
is-conj-with-TF (*FNot* φ) $\longleftrightarrow$ ($\exists$ *v*. φ = *FVar v* $\vee$ φ = *FF* $\vee$ φ = *FT*)
⟨*proof*⟩

lemma *grouped-by-COr-FNot*:
grouped-by COr (*FNot* φ) $\longleftrightarrow$ ($\exists$ *v*. φ = *FVar v* $\vee$ φ = *FF* $\vee$ φ = *FT*)
⟨*proof*⟩

lemma
shows *no-T-F-FF*[*simp*]: $\neg$*no-T-F FF* **and**
no-T-F-FT[*simp*]: $\neg$*no-T-F FT*
⟨*proof*⟩

lemma *grouped-by-CAnd-FAnd*:
grouped-by CAnd (*FAnd* φ*1* φ*2*) $\longleftrightarrow$ *grouped-by CAnd* φ*1* $\wedge$ *grouped-by CAnd* φ*2*
⟨*proof*⟩

lemma *grouped-by-COr-FOr*:
grouped-by COr (*FOr* φ*1* φ*2*) $\longleftrightarrow$ *grouped-by COr* φ*1* $\wedge$ *grouped-by COr* φ*2*
⟨*proof*⟩

lemma *grouped-by-COr-FAnd*[*simp*]: $\neg$ *grouped-by COr* (*FAnd* φ*1* φ*2*)
⟨*proof*⟩

lemma *grouped-by-COr-FEq*[*simp*]: $\neg$ *grouped-by COr* (*FEq* φ*1* φ*2*)
⟨*proof*⟩

lemma [*simp*]: $\neg$*grouped-by COr* (*FImp* φ ψ)
⟨*proof*⟩

lemma [*simp*]: $\neg$ *is-conj-with-TF* (*FImp* φ ψ)
⟨*proof*⟩

lemma [*simp*]: $\neg$*grouped-by COr* (*FEq* φ ψ)
⟨*proof*⟩

lemma [*simp*]: $\neg$ *is-conj-with-TF* (*FEq* φ ψ)
⟨*proof*⟩

lemma *is-conj-with-TF-Fand*:
is-conj-with-TF (*FAnd* $\varphi 1$ $\varphi 2$) $\Longrightarrow$ *is-conj-with-TF* $\varphi 1$ $\wedge$ *is-conj-with-TF* $\varphi 2$
⟨*proof*⟩

lemma *is-conj-with-TF-FOr*:
is-conj-with-TF (*FOr* $\varphi 1$ $\varphi 2$) $\Longrightarrow$ *grouped-by COr* $\varphi 1$ $\wedge$ *grouped-by COr* $\varphi 2$
⟨*proof*⟩

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

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

theorem
fixes φ :: *'v propo*
assumes *is-cnf* φ
shows *eval A* φ $\longleftrightarrow$ *Partial-Clausal-Logic.true-clss* ({*Pos v*|*v*. *A v*} $\cup$ {*Neg v*|*v*. $\neg$*A v*})
(*mset-of-formula* φ)
⟨*proof*⟩

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

begin

Chapter 4

Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

4.1 Resolution

4.1.1 Simplification Rules

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

lemma *simplify-preserves-un-sat'*:
fixes $N\ N' :: 'v\ clauses$
assumes *simplify* $N\ N'$
and *total-over-m* $I\ N$
shows $I \models s\ N' \longrightarrow I \models s\ N$
⟨*proof*⟩

lemma *simplify-preserves-un-sat*:
fixes $N\ N' :: 'v\ clauses$
assumes *simplify* $N\ N'$
and *total-over-m* $I\ N$
shows $I \models s\ N \longrightarrow I \models s\ N'$
⟨*proof*⟩

lemma *simplify-preserves-un-sat''*:
fixes $N\ N' :: 'v\ clauses$
assumes *simplify* $N\ N'$
and *total-over-m* $I\ N'$
shows $I \models s\ N \longrightarrow I \models s\ N'$
⟨*proof*⟩

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

$\langle proof \rangle$

lemma *simplify-preserves-finite*:
assumes *simplify* ψ ψ'
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
$\langle proof \rangle$

lemma *rtranclp-simplify-preserves-finite*:
assumes *rtranclp simplify* ψ ψ'
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
$\langle proof \rangle$

lemma *simplify-atms-of-ms*:
assumes *simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
$\langle proof \rangle$

lemma *rtranclp-simplify-atms-of-ms*:
assumes *rtranclp simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
$\langle proof \rangle$

lemma *factoring-imp-simplify*:
assumes $\{\#L\#\} + \{\#L\#\} + C \in N$
shows $\exists N'.$ *simplify* N N'
$\langle proof \rangle$

4.1.2 Unconstrained Resolution

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

lemma *uncon-res-increasing*:
assumes *uncon-res* S S' **and** $\psi \in S$
shows $\psi \in S'$
$\langle proof \rangle$

lemma *rtranclp-uncon-inference-increasing*:
assumes *rtranclp uncon-res* S S' **and** $\psi \in S$
shows $\psi \in S'$
$\langle proof \rangle$

Subsumption

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

lemma *subsumes-refl*[*simp*]:
subsumes χ χ

$\langle proof \rangle$

lemma *subsumes-subsumption*:
assumes *subsumes* $D\ \chi$
and $C \subset\# D$ **and** $\neg$*tautology* χ
shows *subsumes* $C\ \chi$ $\langle proof \rangle$

lemma *subsumes-tautology*:
assumes *subsumes* $(C + \{\#Pos\ P\#\} + \{\#Neg\ P\#\})\ \chi$
shows *tautology* χ
$\langle proof \rangle$

4.1.3 Inference Rule

type-synonym $'v$ *state* $=$ $'v$ *clauses* $\times$ ($'v$ *clause* $\times$ $'v$ *clause*) *set*
inductive *inference-clause* :: $'v$ *state* $\Rightarrow$ $'v$ *clause* $\times$ ($'v$ *clause* $\times$ $'v$ *clause*) *set* $\Rightarrow$ *bool*
(**infix** $\Rightarrow_{\text{Res}}$ *100*) **where**
resolution:
$\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$ *already-used*
$\Longrightarrow$ *inference-clause* (N, *already-used*) ($C + D$, *already-used* $\cup$ $\{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\})$ |
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$ *inference-clause* (N, *already-used*) ($C + \{\#L\#\}$, *already-used*)

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

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

lemma *inference-clause-preserves-already-used-inv*:
assumes *inference-clause* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* (*fst* $S \cup \{$*fst* $S'\}$, *snd* S')
$\langle proof \rangle$

lemma *inference-preserves-already-used-inv*:
assumes *inference* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
$\langle proof \rangle$

lemma *rtranclp-inference-preserves-already-used-inv*:
assumes *rtranclp inference* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
$\langle proof \rangle$

lemma *subsumes-condensation*:

assumes *subsumes* $(C + \{\#L\#\} + \{\#L\#\})$ D
shows *subsumes* $(C + \{\#L\#\})$ D
⟨*proof*⟩

lemma *simplify-preserves-already-used-inv*:
assumes *simplify* N N'
and *already-used-inv* $(N,$ *already-used*$)$
shows *already-used-inv* $(N',$ *already-used*$)$
⟨*proof*⟩

lemma
factoring-satisfiable: $I \models \{\#L\#\} + \{\#L\#\} + C \longleftrightarrow I \models \{\#L\#\} + C$ **and**
resolution-satisfiable:
consistent-interp $I \Longrightarrow I \models \{\#Pos\ p\#\} + C \Longrightarrow I \models \{\#Neg\ p\#\} + D \Longrightarrow I \models C + D$ **and**
factoring-same-vars: *atms-of* $(\{\#L\#\} + \{\#L\#\} + C) =$ *atms-of* $(\{\#L\#\} + C)$
⟨*proof*⟩

lemma *inference-increasing*:
assumes *inference* S S' **and** $\psi \in$ *fst* S
shows $\psi \in$ *fst* S'
⟨*proof*⟩

lemma *rtranclp-inference-increasing*:
assumes *rtranclp inference* S S' **and** $\psi \in$ *fst* S
shows $\psi \in$ *fst* S'
⟨*proof*⟩

lemma *inference-clause-already-used-increasing*:
assumes *inference-clause* S S'
shows *snd* $S \subseteq$ *snd* S'
⟨*proof*⟩

lemma *inference-already-used-increasing*:
assumes *inference* S S'
shows *snd* $S \subseteq$ *snd* S'
⟨*proof*⟩

lemma *inference-clause-preserves-un-sat*:
fixes N N' :: *'v clauses*
assumes *inference-clause* T T'
and *total-over-m* I $($*fst* $T)$
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $T \longleftrightarrow I \models s$ *fst* $T \cup \{$*fst* $T'\}$
⟨*proof*⟩

lemma *inference-preserves-un-sat*:
fixes N N' :: *'v clauses*
assumes *inference* T T'
and *total-over-m* I $($*fst* $T)$
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $T \longleftrightarrow I \models s$ *fst* T'
⟨*proof*⟩

lemma *inference-clause-preserves-atms-of-ms*:
assumes *inference-clause* S S'
shows *atms-of-ms* (*fst* (*fst* $S \cup \{$*fst* $S'\}$, *snd* S')) $\subseteq$ *atms-of-ms* (*fst* S)
⟨*proof*⟩

lemma *inference-preserves-atms-of-ms*:
fixes N N' :: *'v clauses*
assumes *inference* T T'
shows *atms-of-ms* (*fst* T') $\subseteq$ *atms-of-ms* (*fst* T)
⟨*proof*⟩

lemma *inference-preserves-total*:
fixes N N' :: *'v clauses*
assumes *inference* (N, *already-used*) (N', *already-used'*)
shows *total-over-m* I $N \Longrightarrow$ *total-over-m* I N'
⟨*proof*⟩

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference* T T'
shows *total-over-m* I (*fst* T) $\Longrightarrow$ *total-over-m* I (*fst* T')
⟨*proof*⟩

lemma *rtranclp-inference-preserves-un-sat*:
assumes *rtranclp inference* N N'
and *total-over-m* I (*fst* N)
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $N \longleftrightarrow I \models s$ *fst* N'
⟨*proof*⟩

lemma *inference-preserves-finite*:
assumes *inference* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *inference-preserves-finite-snd*:
assumes *inference* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *consistent-interp-insert*:
assumes *consistent-interp* I
and *atm-of* $P \notin$ *atm-of* ` I

shows *consistent-interp* (*insert P I*)
⟨*proof*⟩

lemma *simplify-clause-preserves-sat*:
assumes *simp*: *simplify* ψ ψ'
and *satisfiable* ψ'
shows *satisfiable* ψ
⟨*proof*⟩

lemma *simplify-preserves-unsat*:
assumes *inference* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *inference-preserves-unsat*:
assumes $inference^{**}$ S S'
shows *satisfiable* (*fst* S') $\longrightarrow$ *satisfiable* (*fst* S)
⟨*proof*⟩

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

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

lemma *sem-tree-size*[*case-names bigger*]:
($\bigwedge$*xs*:: *'v sem-tree*. ($\bigwedge$*ys*:: *'v sem-tree*. *sem-tree-size ys* < *sem-tree-size xs* $\Longrightarrow$ *P ys*) $\Longrightarrow$ *P xs*)
$\Longrightarrow$ *P xs*
⟨*proof*⟩

fun *partial-interps* :: *'v sem-tree* $\Rightarrow$ *'v interp* $\Rightarrow$ *'v clauses* $\Rightarrow$ *bool* **where**
partial-interps Leaf I $\psi = (\exists\chi.\ \neg\ I \models \chi \wedge \chi \in \psi \wedge$ *total-over-m I* $\{\chi\})$ |
partial-interps (*Node v ag ad*) *I* $\psi \longleftrightarrow$
(*partial-interps ag* ($I \cup \{Pos\ v\}$) ψ $\wedge$ *partial-interps ad* ($I\cup \{Neg\ v\}$) ψ)

lemma *simplify-preserve-partial-leaf*:
simplify N N' $\Longrightarrow$ *partial-interps Leaf I N* $\Longrightarrow$ *partial-interps Leaf I* N'
⟨*proof*⟩

lemma *simplify-preserve-partial-tree*:
assumes *simplify* N N'
and *partial-interps t I N*
shows *partial-interps t I* N'
⟨*proof*⟩

lemma *inference-preserve-partial-tree*:
assumes *inference* S S'
and *partial-interps t I* (*fst* S)
shows *partial-interps t I* (*fst* S')
⟨*proof*⟩

lemma *rtranclp-inference-preserve-partial-tree*:
  assumes *rtranclp inference* N N'
  and *partial-interps* t I (*fst* N)
  shows *partial-interps* t I (*fst* N')
  ⟨*proof*⟩

function *build-sem-tree* :: $'v$ :: *linorder set* ⇒ $'v$ *clauses* ⇒ $'v$ *sem-tree* **where**
build-sem-tree atms ψ =
  (*if atms* = {} ∨ ¬ *finite atms*
  then Leaf
  else Node (*Min atms*) (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
    (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ))
⟨*proof*⟩
termination
  ⟨*proof*⟩
declare *build-sem-tree.induct*[*case-names tree*]

lemma *unsatisfiable-empty*[*simp*]:
  ¬*unsatisfiable* {}
  ⟨*proof*⟩

lemma *partial-interps-build-sem-tree-atms-general*:
  fixes ψ :: $'v$ :: *linorder clauses* **and** p :: $'v$ *literal list*
  assumes *unsat*: *unsatisfiable* ψ **and** *finite* ψ **and** *consistent-interp* I
  and *finite atms*
  and *atms-of-ms* ψ = *atms* ∪ *atms-of-s* I **and** *atms* ∩ *atms-of-s* I = {}
  shows *partial-interps* (*build-sem-tree atms* ψ) I ψ
  ⟨*proof*⟩

lemma *partial-interps-build-sem-tree-atms*:
  fixes ψ :: $'v$ :: *linorder clauses* **and** p :: $'v$ *literal list*
  assumes *unsat*: *unsatisfiable* ψ **and** *finite*: *finite* ψ
  shows *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
⟨*proof*⟩

lemma *can-decrease-count*:
  fixes ψ'' :: $'v$ *clauses* × ($'v$ *clause* × $'v$ *clause* × $'v$) *set*
  assumes *count* χ L = n
  and L ∈# χ **and** χ ∈ *fst* ψ
  shows $\exists\psi'$ χ'. *inference*** ψ ψ' ∧ χ' ∈ *fst* ψ' ∧ (∀ L. L ∈# χ ⟷ L ∈# χ')
        ∧ *count* χ' L = *1*
        ∧ (∀ φ. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ')
        ∧ (I ⊨ χ ⟷ I ⊨ χ')
        ∧ (∀ I'. *total-over-m* I' {χ} ⟶ *total-over-m* I' {χ'})
  ⟨*proof*⟩

lemma *can-decrease-tree-size*:
  fixes ψ :: $'v$ *state* **and** *tree* :: $'v$ *sem-tree*
  assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
  and *partial-interps tree* I (*fst* ψ)
  shows ∃(*tree'*:: $'v$ *sem-tree*) ψ'. *inference*** ψ ψ' ∧ *partial-interps tree'* I (*fst* ψ')
      ∧ (*sem-tree-size tree'* < *sem-tree-size tree* ∨ *sem-tree-size tree* = *0*)
  ⟨*proof*⟩

lemma *inference-completeness-inv*:
  fixes ψ :: $'v$ ::*linorder state*
  assumes
    unsat: $\neg$*satisfiable* (*fst* ψ) **and**
    finite: *finite* (*fst* ψ) **and**
    a-u-v: *already-used-inv* ψ
  shows $\exists\,\psi'$. (*inference*** ψ ψ' $\wedge$ $\{\#\}$ $\in$ *fst* ψ')
$\langle$*proof*$\rangle$

lemma *inference-completeness*:
  fixes ψ :: $'v$ ::*linorder state*
  assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
  and *finite*: *finite* (*fst* ψ)
  and *snd* ψ = {}
  shows $\exists\,\psi'$. (*rtranclp inference* ψ ψ' $\wedge$ $\{\#\}$ $\in$ *fst* ψ')
$\langle$*proof*$\rangle$

lemma *inference-soundness*:
  fixes ψ :: $'v$ ::*linorder state*
  assumes *rtranclp inference* ψ ψ' **and** $\{\#\}$ $\in$ *fst* ψ'
  shows *unsatisfiable* (*fst* ψ)
  $\langle$*proof*$\rangle$

lemma *inference-soundness-and-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows ($\exists\,\psi'$. (*inference*** ψ ψ' $\wedge$ $\{\#\}$ $\in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
  $\langle$*proof*$\rangle$

4.1.4 Lemma about the simplified state

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

lemma *simplified-count*:
  assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
  shows *count* χ $L \leq 1$
$\langle$*proof*$\rangle$

lemma *simplified-no-both*:
  assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
  shows $\neg$ ($L \in\# \chi \wedge -L \in\# \chi$)
$\langle$*proof*$\rangle$

lemma *simplified-not-tautology*:
  assumes *simplified* $\{\psi\}$
  shows ~*tautology* ψ
$\langle$*proof*$\rangle$

lemma *simplified-remove*:
  assumes *simplified* $\{\psi\}$
  shows *simplified* $\{\psi - \{\#l\#\}\}$
$\langle$*proof*$\rangle$

lemma *in-simplified-simplified*:

assumes *simp*: *simplified* ψ **and** *incl*: $\psi' \subseteq \psi$
shows *simplified* ψ'
⟨*proof*⟩

lemma *simplified-in*:
assumes *simplified* ψ
and $N \in \psi$
shows *simplified* $\{N\}$
⟨*proof*⟩

lemma *subsumes-imp-formula*:
assumes $\psi \leq\# \varphi$
shows $\{\psi\} \models p\ \varphi$
⟨*proof*⟩

lemma *simplified-imp-distinct-mset-tauto*:
assumes *simp*: *simplified* ψ'
shows *distinct-mset-set* ψ' **and** $\forall \chi \in \psi'. \neg$*tautology* χ
⟨*proof*⟩

lemma *simplified-no-more-full1-simplified*:
assumes *simplified* ψ
shows $\neg$*full1 simplify* ψ ψ'
⟨*proof*⟩

4.1.5 Resolution and Invariants

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

Invariants

lemma *resolution-finite*:
assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *rtranclp-resolution-finite*:
assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *resolution-finite-snd*:
assumes *resolution* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *rtranclp-resolution-finite-snd*:
assumes *resolution*** ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *resolution-always-simplified*:
assumes *resolution* ψ ψ'

shows *simplified* (*fst* ψ')
⟨*proof*⟩

lemma *tranclp-resolution-always-simplified*:
assumes *tranclp resolution* ψ ψ'
shows *simplified* (*fst* ψ')
⟨*proof*⟩

lemma *resolution-atms-of*:
assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-resolution-atms-of*:
assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
⟨*proof*⟩

lemma *resolution-include*:
assumes *res*: *resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
⟨*proof*⟩

lemma *rtranclp-resolution-include*:
assumes *res*: *tranclp resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
⟨*proof*⟩

abbreviation *already-used-all-simple*
:: (*'a literal multiset* $\times$ *'a literal multiset*) *set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool* **where**
already-used-all-simple already-used vars $\equiv$
($\forall$ (*A*, *B*) $\in$ *already-used*. *simplified* {*A*} $\wedge$ *simplified* {*B*} $\wedge$ *atms-of A* $\subseteq$ *vars* $\wedge$ *atms-of B* $\subseteq$ *vars*)

lemma *already-used-all-simple-vars-incl*:
assumes *vars* $\subseteq$ *vars'*
shows *already-used-all-simple a vars* $\Longrightarrow$ *already-used-all-simple a vars'*
⟨*proof*⟩

lemma *inference-clause-preserves-already-used-all-simple*:
assumes *inference-clause S S'*
and *already-used-all-simple* (*snd S*) *vars*
and *simplified* (*fst S*)
and *atms-of-ms* (*fst S*) $\subseteq$ *vars*
shows *already-used-all-simple* (*snd* (*fst S* $\cup$ {*fst S'*}, *snd S'*)) *vars*
⟨*proof*⟩

lemma *inference-preserves-already-used-all-simple*:
assumes *inference S S'*
and *already-used-all-simple* (*snd S*) *vars*
and *simplified* (*fst S*)
and *atms-of-ms* (*fst S*) $\subseteq$ *vars*
shows *already-used-all-simple* (*snd S'*) *vars*
⟨*proof*⟩

lemma *already-used-all-simple-inv*:
assumes *resolution S S'*

and *already-used-all-simple* (*snd S*) *vars*
and *atms-of-ms* (*fst S*) ⊆ *vars*
shows *already-used-all-simple* (*snd S′*) *vars*
⟨*proof*⟩

lemma *rtranclp-already-used-all-simple-inv*:
assumes *resolution*** *S S′*
and *already-used-all-simple* (*snd S*) *vars*
and *atms-of-ms* (*fst S*) ⊆ *vars*
and *finite* (*fst S*)
shows *already-used-all-simple* (*snd S′*) *vars*
⟨*proof*⟩

lemma *inference-clause-simplified-already-used-subset*:
assumes *inference-clause S S′*
and *simplified* (*fst S*)
shows *snd S* ⊂ *snd S′*
⟨*proof*⟩

lemma *inference-simplified-already-used-subset*:
assumes *inference S S′*
and *simplified* (*fst S*)
shows *snd S* ⊂ *snd S′*
⟨*proof*⟩

lemma *resolution-simplified-already-used-subset*:
assumes *resolution S S′*
and *simplified* (*fst S*)
shows *snd S* ⊂ *snd S′*
⟨*proof*⟩

lemma *tranclp-resolution-simplified-already-used-subset*:
assumes *tranclp resolution S S′*
and *simplified* (*fst S*)
shows *snd S* ⊂ *snd S′*
⟨*proof*⟩

abbreviation *already-used-top vars* ≡ *simple-clss vars* × *simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *s* ⊆ *already-used-top vars*
⟨*proof*⟩

lemma *already-used-top-finite*:
assumes *finite vars*
shows *finite* (*already-used-top vars*)
⟨*proof*⟩

lemma *already-used-top-increasing*:
assumes *var* ⊆ *var′* **and** *finite var′*
shows *already-used-top var* ⊆ *already-used-top var′*
⟨*proof*⟩

lemma *already-used-all-simple-finite*:
fixes *s* :: (*′a literal multiset* × *′a literal multiset*) *set* **and** *vars* :: *′a set*

assumes *already-used-all-simple s vars* **and** *finite vars*
shows *finite s*
⟨*proof*⟩

abbreviation *card-simple vars* $\psi \equiv$ *card* (*already-used-top vars* $- \psi$)

lemma *resolution-card-simple-decreasing*:
assumes *res*: *resolution* $\psi\ \psi'$
and *a-u-s*: *already-used-all-simple* (*snd* ψ) *vars*
and *finite-v*: *finite vars*
and *finite-fst*: *finite* (*fst* ψ)
and *finite-snd*: *finite* (*snd* ψ)
and *simp*: *simplified* (*fst* ψ)
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing*:
assumes *tranclp resolution* $\psi\ \psi'$ **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
and *finite-v*: *finite vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing-2*:
assumes *tranclp resolution* $\psi\ \psi'$
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* $\psi = \{\}$
and *simplified* (*fst* ψ)
shows *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ') $<$ *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ)
⟨*proof*⟩

well-foundness if the relation

lemma *wf-simplified-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: ′*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ *simplified* (*fst x*)
$\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}
⟨*proof*⟩

lemma *wf-simplified-resolution′*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: ′*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ $\neg$*simplified* (*fst x*)
$\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}
⟨*proof*⟩

lemma *wf-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* ({(*y*:: ′*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ *simplified* (*fst x*)
$\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}
$\cup$ {(*y*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ $\neg$ *simplified* (*fst x*) $\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*)

$\wedge$ *already-used-all-simple* (*snd* x) *vars*) $\wedge$ *resolution* x y}) (**is** *wf* (*?R* $\cup$ *?S*))
⟨*proof*⟩

lemma *rtrancp-simplify-already-used-inv*:
assumes *simplify*** S S'
and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
⟨*proof*⟩

lemma *full1-simplify-already-used-inv*:
assumes *full1 simplify* S S'
and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
⟨*proof*⟩

lemma *full-simplify-already-used-inv*:
assumes *full simplify* S S'
and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
⟨*proof*⟩
lemma *resolution-already-used-inv*:
assumes *resolution* S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *rtranclp-resolution-already-used-inv*:
assumes *resolution*** S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *rtanclp-simplify-preserves-unsat*:
assumes *simplify*** ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
⟨*proof*⟩

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
⟨*proof*⟩

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
⟨*proof*⟩

lemma *resolution-preserves-unsat*:
assumes *resolution* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-unsat*:
assumes *resolution*** ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-simplify-preserve-partial-tree*:
  assumes $simplify^{**}\ N\ N'$
  and *partial-interps t I N*
  shows *partial-interps t I N′*
  ⟨*proof*⟩

lemma *full1-simplify-preserve-partial-tree*:
  assumes *full1 simplify N N′*
  and *partial-interps t I N*
  shows *partial-interps t I N′*
  ⟨*proof*⟩

lemma *full-simplify-preserve-partial-tree*:
  assumes *full simplify N N′*
  and *partial-interps t I N*
  shows *partial-interps t I N′*
  ⟨*proof*⟩

lemma *resolution-preserve-partial-tree*:
  assumes *resolution S S′*
  and *partial-interps t I* (*fst S*)
  shows *partial-interps t I* (*fst S′*)
  ⟨*proof*⟩

lemma *rtranclp-resolution-preserve-partial-tree*:
  assumes $resolution^{**}\ S\ S'$
  and *partial-interps t I* (*fst S*)
  shows *partial-interps t I* (*fst S′*)
  ⟨*proof*⟩
  thm *nat-less-induct nat.induct*

lemma *nat-ge-induct*[*case-names 0 Suc*]:
  assumes *P 0*
  and $\bigwedge n.\ (\bigwedge m.\ m<Suc\ n \Longrightarrow P\ m) \Longrightarrow P\ (Suc\ n)$
  shows *P n*
  ⟨*proof*⟩

lemma *wf-always-more-step-False*:
  assumes *wf R*
  shows $(\forall x.\ \exists z.\ (z,\ x)\in R) \Longrightarrow False$
 ⟨*proof*⟩

lemma *finite-finite-mset-element-of-mset*[*simp*]:
  assumes *finite N*
  shows *finite* $\{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\}$
  ⟨*proof*⟩

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

interpretation *sum-count-ge-2*:
  $folding\ (\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0$
rewrites

folding.F ($\lambda\varphi$. *op* +(*msetsum* {#*count* φ *L* |*L* ∈# φ. *2* ≤ *count* φ *L*#})) *0* = *sum-count-ge-2*
⟨*proof*⟩

lemma *finite-incl-le-setsum*:
finite (*B*::′*a multiset set*) $\Longrightarrow$ *A* ⊆ *B* $\Longrightarrow$ Ξ *A* ≤ Ξ *B*
⟨*proof*⟩

lemma *simplify-finite-measure-decrease*:
simplify N N′ $\Longrightarrow$ *finite N* $\Longrightarrow$ *card N′* + Ξ *N′* < *card N* + Ξ *N*
⟨*proof*⟩

lemma *simplify-terminates*:
wf {(*N′*, *N*). *finite N* ∧ *simplify N N′*}
⟨*proof*⟩

lemma *wf-terminates*:
assumes *wf r*
shows ∃ *N′*.(*N′*, *N*)∈ r^* ∧ (∀ *N″*. (*N″*, *N′*)∉ *r*)
⟨*proof*⟩

lemma *rtranclp-simplify-terminates*:
assumes *fin*: *finite N*
shows ∃ *N′*. $simplify^{**}$ *N N′* ∧ *simplified N′*
⟨*proof*⟩

lemma *finite-simplified-full1-simp*:
assumes *finite N*
shows *simplified N* ∨ (∃ *N′*. *full1 simplify N N′*)
⟨*proof*⟩

lemma *finite-simplified-full-simp*:
assumes *finite N*
shows ∃ *N′*. *full simplify N N′*
⟨*proof*⟩

lemma *can-decrease-tree-size-resolution*:
fixes ψ :: ′*v state* **and** *tree* :: ′*v sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
and *simplified* (*fst* ψ)
shows ∃(*tree′*:: ′*v sem-tree*) ψ'. $resolution^{**}$ ψ ψ' ∧ *partial-interps tree′ I* (*fst* ψ')
∧ (*sem-tree-size tree′* < *sem-tree-size tree* ∨ *sem-tree-size tree* = *0*)
⟨*proof*⟩

lemma *resolution-completeness-inv*:
fixes ψ :: ′*v* ::*linorder state*
assumes
unsat: ¬*satisfiable* (*fst* ψ) **and**
finite: *finite* (*fst* ψ) **and**
a-u-v: *already-used-inv* ψ
shows ∃ ψ'. ($resolution^{**}$ ψ ψ' ∧ {#} ∈ *fst* ψ')
⟨*proof*⟩

lemma *resolution-preserves-already-used-inv*:
assumes *resolution S S′*

and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-already-used-inv*:
assumes $resolution^{**}$ S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *resolution-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* $\psi = \{\}$
shows $\exists \psi'.$ ($resolution^{**}$ ψ ψ' $\wedge$ $\{\#\} \in$ *fst* ψ')
⟨*proof*⟩

lemma *rtranclp-preserves-sat*:
assumes $simplify^{**}$ S S'
and *satisfiable* S
shows *satisfiable* S'
⟨*proof*⟩

lemma *resolution-preserves-sat*:
assumes *resolution* S S'
and *satisfiable* (*fst* S)
shows *satisfiable* (*fst* S')
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-sat*:
assumes $resolution^{**}$ S S'
and *satisfiable* (*fst* S)
shows *satisfiable* (*fst* S')
⟨*proof*⟩

lemma *resolution-soundness*:
fixes ψ :: $'v$ ::*linorder state*
assumes $resolution^{**}$ ψ ψ' **and** $\{\#\} \in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *resolution-soundness-and-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd*: *snd* $\psi = \{\}$
shows ($\exists \psi'.$ ($resolution^{**}$ ψ ψ' $\wedge$ $\{\#\} \in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *simplified-falsity*:
assumes *simp*: *simplified* ψ
and $\{\#\} \in \psi$
shows $\psi = \{\{\#\}\}$
⟨*proof*⟩

lemma *simplify-falsity-in-preserved*:
  assumes *simplify* χs $\chi s'$
  and $\{\#\} \in \chi s$
  shows $\{\#\} \in \chi s'$
  ⟨*proof*⟩

lemma *rtranclp-simplify-falsity-in-preserved*:
  assumes *simplify*** χs $\chi s'$
  and $\{\#\} \in \chi s$
  shows $\{\#\} \in \chi s'$
  ⟨*proof*⟩

lemma *resolution-falsity-get-falsity-alone*:
  assumes *finite* (*fst* ψ)
  shows ($\exists\, \psi'$. (*resolution*** ψ ψ' $\wedge$ $\{\#\} \in$ *fst* ψ')) $\longleftrightarrow$ ($\exists$ *a-u-v*. *resolution*** ψ ($\{\{\#\}\}$, *a-u-v*))
    (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *resolution-soundness-and-completeness'*:
  fixes ψ :: *'v* ::*linorder state*
  assumes
    finite: *finite* (*fst* ψ)**and**
    snd: *snd* ψ = {}
  shows ($\exists$ *a-u-v*. (*resolution*** ψ ($\{\{\#\}\}$, *a-u-v*))) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
    ⟨*proof*⟩

end
theory *Prop-Superposition*
imports *Partial-Clausal-Logic ../lib/Herbrand-Interpretation*
begin

4.2 Superposition

no-notation *Herbrand-Interpretation.true-cls* (**infix** $\models$ *50*)
notation *Herbrand-Interpretation.true-cls* (**infix** $\models$*h 50*)

no-notation *Herbrand-Interpretation.true-clss* (**infix** $\models$*s 50*)
notation *Herbrand-Interpretation.true-clss* (**infix** $\models$*hs 50*)

lemma *herbrand-interp-iff-partial-interp-cls*:
  $S \models h\ C \longleftrightarrow \{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\} \models C$
  ⟨*proof*⟩

lemma *herbrand-consistent-interp*:
  consistent-interp ($\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\}$)
  ⟨*proof*⟩

lemma *herbrand-total-over-set*:
  total-over-set ($\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\}$) *T*
  ⟨*proof*⟩

lemma *herbrand-total-over-m*:
  total-over-m ($\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\}$) *T*
  ⟨*proof*⟩

lemma *herbrand-interp-iff-partial-interp-clss*:
  $S \models hs\ C \longleftrightarrow \{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\} \models s\ C$
  ⟨*proof*⟩

definition *clss-lt* :: *'a::wellorder clauses* ⇒ *'a clause* ⇒ *'a clauses* **where**
clss-lt N C = $\{D \in N.\ D\ \#{\subset}\#\ C\}$

notation (*latex* **output**)
 clss-lt (-<^bsup>-<^esup>)

locale *selection* =
  fixes *S* :: *'a clause* ⇒ *'a clause*
  assumes
    S-selects-subseteq: $\bigwedge C.\ S\ C \leq\#\ C$ **and**
    S-selects-neg-lits: $\bigwedge C\ L.\ L \in\#\ S\ C \Longrightarrow$ *is-neg L*

locale *ground-resolution-with-selection* =
  selection S **for** *S* :: (*'a* :: *wellorder*) *clause* ⇒ *'a clause*
begin

context
  fixes *N* :: *'a clause set*
begin

We do not create an equivalent of δ, but we directly defined N_C by inlining the definition.

function
  production :: *'a clause* ⇒ *'a interp*
where
  production C =
   $\{A.\ C \in N \land C \neq \{\#\} \land Max\ (set\text{-}mset\ C) = Pos\ A \land count\ C\ (Pos\ A) \leq 1$
     $\land \neg (\bigcup D \in \{D.\ D\ \#{\subset}\#\ C\}.\ production\ D) \models h\ C \land S\ C = \{\#\}\}$
  ⟨*proof*⟩
termination ⟨*proof*⟩

declare *production.simps*[*simp del*]

definition *interp* :: *'a clause* ⇒ *'a interp* **where**
  interp C = $(\bigcup D \in \{D.\ D\ \#{\subset}\#\ C\}.\ production\ D)$

lemma *production-unfold*:
  $production\ C = \{A.\ C \in N \land C \neq \{\#\} \land Max\ (set\text{-}mset\ C) = Pos\ A \land count\ C\ (Pos\ A) \leq 1 \land \neg$ $interp\ C \models h\ C \land S\ C = \{\#\}\}$
  ⟨*proof*⟩

abbreviation *productive A* ≡ (*production A* ≠ {})

abbreviation *produces* :: *'a clause* ⇒ *'a* ⇒ *bool* **where**
  produces C A ≡ *production C* = {*A*}

lemma *producesD*:
  $produces\ C\ A \Longrightarrow C \in N \land C \neq \{\#\} \land Pos\ A = Max\ (set\text{-}mset\ C) \land count\ C\ (Pos\ A) \leq 1 \land$
    $\neg\ interp\ C \models h\ C \land S\ C = \{\#\}$
  ⟨*proof*⟩

lemma $produces\ C\ A \Longrightarrow Pos\ A \in\#\ C$
  ⟨*proof*⟩

lemma *interp'-def-in-set*:
$interp\ C = (\bigcup D \in \{D \in N.\ D \#\subset\# \ C\}.\ production\ D)$
⟨*proof*⟩

lemma *production-iff-produces*:
$produces\ D\ A \longleftrightarrow A \in production\ D$
⟨*proof*⟩

definition $Interp :: 'a\ clause \Rightarrow 'a\ interp$ **where**
$Interp\ C = interp\ C \cup production\ C$

lemma
assumes $produces\ C\ P$
shows $Interp\ C \models h\ C$
⟨*proof*⟩

definition $INTERP :: 'a\ interp$ **where**
$INTERP = (\bigcup D \in N.\ production\ D)$

lemma *interp-subseteq-Interp*[*simp*]: $interp\ C \subseteq Interp\ C$
⟨*proof*⟩

lemma *Interp-as-UNION*: $Interp\ C = (\bigcup D \in \{D.\ D \#\subseteq\# \ C\}.\ production\ D)$
⟨*proof*⟩

lemma *productive-not-empty*: $productive\ C \Longrightarrow C \neq \{\#\}$
⟨*proof*⟩

lemma *productive-imp-produces-Max-literal*: $productive\ C \Longrightarrow produces\ C\ (atm\text{-}of\ (Max\ (set\text{-}mset\ C)))$
⟨*proof*⟩

lemma *productive-imp-produces-Max-atom*: $productive\ C \Longrightarrow produces\ C\ (Max\ (atms\text{-}of\ C))$
⟨*proof*⟩

lemma *produces-imp-Max-literal*: $produces\ C\ A \Longrightarrow A = atm\text{-}of\ (Max\ (set\text{-}mset\ C))$
⟨*proof*⟩

lemma *produces-imp-Max-atom*: $produces\ C\ A \Longrightarrow A = Max\ (atms\text{-}of\ C)$
⟨*proof*⟩

lemma *produces-imp-Pos-in-lits*: $produces\ C\ A \Longrightarrow Pos\ A \in\# \ C$
⟨*proof*⟩

lemma *productive-in-N*: $productive\ C \Longrightarrow C \in N$
⟨*proof*⟩

lemma *produces-imp-atms-leq*: $produces\ C\ A \Longrightarrow B \in atms\text{-}of\ C \Longrightarrow B \leq A$
⟨*proof*⟩

lemma *produces-imp-neg-notin-lits*: $produces\ C\ A \Longrightarrow Neg\ A \notin\# \ C$
⟨*proof*⟩

lemma *less-eq-imp-interp-subseteq-interp*: $C \#\subseteq\# \ D \Longrightarrow interp\ C \subseteq interp\ D$
⟨*proof*⟩

lemma *less-eq-imp-interp-subseteq-Interp*: $C \; \# {\subseteq} \# \; D \Longrightarrow interp \; C \subseteq Interp \; D$
⟨*proof*⟩

lemma *less-imp-production-subseteq-interp*: $C \; \# {\subset} \# \; D \Longrightarrow production \; C \subseteq interp \; D$
⟨*proof*⟩

lemma *less-eq-imp-production-subseteq-Interp*: $C \; \# {\subseteq} \# \; D \Longrightarrow production \; C \subseteq Interp \; D$
⟨*proof*⟩

lemma *less-imp-Interp-subseteq-interp*: $C \; \# {\subset} \# \; D \Longrightarrow Interp \; C \subseteq interp \; D$
⟨*proof*⟩

lemma *less-eq-imp-Interp-subseteq-Interp*: $C \; \# {\subseteq} \# \; D \Longrightarrow Interp \; C \subseteq Interp \; D$
⟨*proof*⟩

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin Interp \; C \Longrightarrow A \in interp \; D \Longrightarrow C \; \# {\subset} \# \; D$
⟨*proof*⟩

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin interp \; C \Longrightarrow A \in interp \; D \Longrightarrow C \; \# {\subset} \# \; D$
⟨*proof*⟩

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin Interp \; C \Longrightarrow A \in Interp \; D \Longrightarrow C \; \# {\subset} \# \; D$
⟨*proof*⟩

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin interp \; C \Longrightarrow A \in Interp \; D \Longrightarrow C \; \# {\subseteq} \# \; D$
⟨*proof*⟩

lemma *interp-subseteq-INTERP*: $interp \; C \subseteq INTERP$
⟨*proof*⟩

lemma *production-subseteq-INTERP*: $production \; C \subseteq INTERP$
⟨*proof*⟩

lemma *Interp-subseteq-INTERP*: $Interp \; C \subseteq INTERP$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: $Neg \; A \in \# \; C$ **and** *d*: $produces \; D \; A$
shows $A \in interp \; C$
⟨*proof*⟩

lemma *neg-notin-Interp-not-produce*: $Neg \; A \in \# \; C \Longrightarrow A \notin Interp \; D \Longrightarrow C \; \# {\subseteq} \# \; D \Longrightarrow \neg \; produces \; D'' \; A$
⟨*proof*⟩

lemma *in-production-imp-produces*: $A \in production \; C \Longrightarrow produces \; C \; A$
⟨*proof*⟩

lemma *not-produces-imp-notin-production*: $\neg \; produces \; C \; A \Longrightarrow A \notin production \; C$
⟨*proof*⟩

lemma *not-produces-imp-notin-interp*: $(\bigwedge D. \; \neg \; produces \; D \; A) \Longrightarrow A \notin interp \; C$
⟨*proof*⟩

The results below corresponds to Lemma 3.4.

Nitpicking: If $D = D'$ and D is productive, $I^D \subseteq I_{D'}$ does not hold.

lemma *true-Interp-imp-general*:
  assumes
    c-le-d: $C\ \#\subseteq\#\ D$ **and**
    d-lt-d': $D\ \#\subset\#\ D'$ **and**
    c-at-d: *Interp* $D \models h\ C$ **and**
    subs: *interp* $D' \subseteq (\bigcup C \in CC.\ production\ C)$
  shows $(\bigcup C \in CC.\ production\ C) \models h\ C$
⟨*proof*⟩

lemma *true-Interp-imp-interp*: $C\ \#\subseteq\#\ D \Longrightarrow D\ \#\subset\#\ D' \Longrightarrow$ *Interp* $D \models h\ C \Longrightarrow$ *interp* $D' \models h\ C$
  ⟨*proof*⟩

lemma *true-Interp-imp-Interp*: $C\ \#\subseteq\#\ D \Longrightarrow D\ \#\subset\#\ D' \Longrightarrow$ *Interp* $D \models h\ C \Longrightarrow$ *Interp* $D' \models h\ C$
  ⟨*proof*⟩

lemma *true-Interp-imp-INTERP*: $C\ \#\subseteq\#\ D \Longrightarrow$ *Interp* $D \models h\ C \Longrightarrow$ *INTERP* $\models h\ C$
  ⟨*proof*⟩

lemma *true-interp-imp-general*:
  assumes
    c-le-d: $C\ \#\subseteq\#\ D$ **and**
    d-lt-d': $D\ \#\subset\#\ D'$ **and**
    c-at-d: *interp* $D \models h\ C$ **and**
    subs: *interp* $D' \subseteq (\bigcup C \in CC.\ production\ C)$
  shows $(\bigcup C \in CC.\ production\ C) \models h\ C$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *true-interp-imp-interp*: $C\ \#\subseteq\#\ D \Longrightarrow D\ \#\subset\#\ D' \Longrightarrow$ *interp* $D \models h\ C \Longrightarrow$ *interp* $D' \models h\ C$
  ⟨*proof*⟩

lemma *true-interp-imp-Interp*: $C\ \#\subseteq\#\ D \Longrightarrow D\ \#\subset\#\ D' \Longrightarrow$ *interp* $D \models h\ C \Longrightarrow$ *Interp* $D' \models h\ C$
  ⟨*proof*⟩

lemma *true-interp-imp-INTERP*: $C\ \#\subseteq\#\ D \Longrightarrow$ *interp* $D \models h\ C \Longrightarrow$ *INTERP* $\models h\ C$
  ⟨*proof*⟩

lemma *productive-imp-false-interp*: *productive* $C \Longrightarrow \neg$ *interp* $C \models h\ C$
  ⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:
  assumes D: $\{\#Pos\ P,\ Pos\ P\#\}\ \#\subset\#\ C$
  shows $\neg$*produces* $C\ P$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma
  assumes D: $C+\{\#Neg\ P\#\}\ \#\subset\#\ D$
  shows *production* $D \neq \{P\}$

⟨*proof*⟩

lemma *in-interp-is-produced*:
 assumes $P \in INTERP$
 shows $\exists D.\ D + \{\#Pos\ P\#\} \in N \wedge produces\ (D + \{\#Pos\ P\#\})\ P$
 ⟨*proof*⟩

end
end

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

4.2.1 We can now define the rules of the calculus

inductive *superposition-rules* :: $'a\ clause \Rightarrow 'a\ clause \Rightarrow 'a\ clause \Rightarrow bool$ **where**
factoring: $superposition\text{-}rules\ (C + \{\#Pos\ P\#\} + \{\#Pos\ P\#\})\ B\ (C + \{\#Pos\ P\#\})$ |
superposition-l: $superposition\text{-}rules\ (C_1 + \{\#Pos\ P\#\})\ (C_2 + \{\#Neg\ P\#\})\ (C_1 + C_2)$

inductive *superposition* :: $'a\ clauses \Rightarrow 'a\ clauses \Rightarrow bool$ **where**
superposition: $A \in N \Longrightarrow B \in N \Longrightarrow superposition\text{-}rules\ A\ B\ C$
 $\Longrightarrow superposition\ N\ (N \cup \{C\})$

definition *abstract-red* :: $'a{::}wellorder\ clause \Rightarrow 'a\ clauses \Rightarrow bool$ **where**
$abstract\text{-}red\ C\ N = (clss\text{-}lt\ N\ C \models p\ C)$

lemma *less-multiset*[*iff*]: $M < N \longleftrightarrow M\ \#{\subset}\#\ N$
 ⟨*proof*⟩

lemma *less-eq-multiset*[*iff*]: $M \leq N \longleftrightarrow M\ \#{\subseteq}\#\ N$
 ⟨*proof*⟩

lemma *herbrand-true-clss-true-clss-cls-herbrand-true-clss*:
 assumes
  AB: $A \models hs\ B$ **and**
  BC: $B \models p\ C$
 shows $A \models h\ C$
⟨*proof*⟩

lemma *abstract-red-subset-mset-abstract-red*:
 assumes
  abstr: *abstract-red C N* **and**
  c-lt-d: $C \subseteq\#\ D$
 shows *abstract-red D N*
⟨*proof*⟩

lemma *true-clss-cls-extended*:
 assumes
  $A \models p\ B$ **and**
  tot: *total-over-m I A* **and**
  cons: *consistent-interp I* **and**
  I-A: $I \models s\ A$
 shows $I \models B$
⟨*proof*⟩
lemma

assumes
CP: ¬ *clss-lt* N ({#C#} + {#E#}) $\models p$ {#C#} + {#*Neg* P#} **and**
clss-lt N ({#C#} + {#E#}) $\models p$ {#E#} + {#*Pos* P#} ∨ *clss-lt* N ({#C#} + {#E#}) $\models p$ {#C#} + {#*Neg* P#}
shows *clss-lt* N ({#C#} + {#E#}) $\models p$ {#E#} + {#*Pos* P#}

⟨*proof*⟩

locale *ground-ordered-resolution-with-redundancy* =
ground-resolution-with-selection +
fixes *redundant* :: ′*a*::*wellorder clause* ⇒ ′*a clauses* ⇒ *bool*
assumes
redundant-iff-abstract: *redundant* A N ⟷ *abstract-red* A N
begin
definition *saturated* :: ′*a clauses* ⇒ *bool* **where**
saturated N ⟷ (∀ A B C. $A \in N$ ⟶ $B \in N$ ⟶ ¬*redundant* A N ⟶ ¬*redundant* B N
⟶ *superposition-rules* A B C ⟶ *redundant* C N ∨ $C \in N$)

lemma
assumes
saturated: *saturated* N **and**
finite: *finite* N **and**
empty: {#} ∉ N
shows *INTERP* N $\models hs$ N
⟨*proof*⟩

end

lemma *tautology-is-redundant*:
assumes *tautology* C
shows *abstract-red* C N
⟨*proof*⟩

lemma *subsumed-is-redundant*:
assumes AB: $A \subset\# B$
and AN: $A \in N$
shows *abstract-red* B N
⟨*proof*⟩

inductive *redundant* :: ′*a clause* ⇒ ′*a clauses* ⇒ *bool* **where**
subsumption: $A \in N \Longrightarrow A \subset\# B \Longrightarrow$ *redundant* B N

lemma *redundant-is-redundancy-criterion*:
fixes A :: ′*a* :: *wellorder clause* **and** N :: ′*a* :: *wellorder clauses*
assumes *redundant* A N
shows *abstract-red* A N
⟨*proof*⟩

lemma *redundant-mono*:
redundant A $N \Longrightarrow A \subseteq\# B \Longrightarrow$ *redundant* B N
⟨*proof*⟩

locale *truc* =
selection S **for** S :: *nat clause* ⇒ *nat clause*
begin

end

end

4.3 Partial Clausal Logic

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

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

begin

4.3.1 Decided Literals

Definition

datatype ('*v*, '*mark*) *ann-lit* =
  is-decided: *Decided* (*lit-of*: '*v literal*) |
  is-proped: *Propagated* (*lit-of*: '*v literal*) (*mark-of*: '*mark*)

lemma *ann-lit-list-induct*[*case-names Nil Decided Propagated*]:
  assumes P [] **and**
  $\bigwedge L\ xs.\ P\ xs \Longrightarrow P\ (Decided\ L\ \#\ xs)$ **and**
  $\bigwedge L\ m\ xs.\ P\ xs \Longrightarrow P\ (Propagated\ L\ m\ \#\ xs)$
  shows $P\ xs$
  ⟨*proof*⟩

lemma *is-decided-ex-Decided*:
  $is\text{-}decided\ L \Longrightarrow (\bigwedge K.\ L = Decided\ K \Longrightarrow P) \Longrightarrow P$
  ⟨*proof*⟩

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

definition *lits-of* :: ('*a*, '*b*) *ann-lit set* ⇒ '*a literal set* **where**
lits-of Ls = *lit-of* ‘ *Ls*

abbreviation *lits-of-l* :: ('*a*, '*b*) *ann-lits* ⇒ '*a literal set* **where**
lits-of-l Ls ≡ *lits-of* (*set Ls*)

lemma *lits-of-l-empty*[*simp*]:
  lits-of {} = {}
  ⟨*proof*⟩

lemma *lits-of-insert*[*simp*]:
  lits-of (*insert L Ls*) = *insert* (*lit-of L*) (*lits-of Ls*)
  ⟨*proof*⟩

lemma *lits-of-l-Un*[*simp*]:
  lits-of ($l \cup l'$) = *lits-of* $l \cup$ *lits-of* l'
  ⟨*proof*⟩

lemma *finite-lits-of-def*[*simp*]:
  finite (*lits-of-l L*)
  ⟨*proof*⟩

abbreviation *unmark* **where**
unmark $\equiv$ (λa. {#*lit-of a*#})

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

abbreviation *unmark-l* **where**
unmark-l M $\equiv$ *unmark-s* (*set M*)

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of*[*simp*]:
atms-of-ms (*unmark-l M'*) = *atm-of* ' *lits-of-l M'*
⟨*proof*⟩

lemma *lits-of-l-empty-is-empty*[*iff*]:
lits-of-l M = {} $\longleftrightarrow$ *M* = []
⟨*proof*⟩

Entailment

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

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

lemma *true-annot-empty-model*[*simp*]:
$\neg$[] $\models$*a* ψ
⟨*proof*⟩

lemma *true-annot-empty*[*simp*]:
$\neg$*I* $\models$*a* {#}
⟨*proof*⟩

lemma *empty-true-annots-def*[*iff*]:
[] $\models$*as* ψ $\longleftrightarrow$ ψ = {}
⟨*proof*⟩

lemma *true-annots-empty*[*simp*]:
I $\models$*as* {}
⟨*proof*⟩

lemma *true-annots-single-true-annot*[*iff*]:
I $\models$*as* {*C*} $\longleftrightarrow$ *I* $\models$*a C*
⟨*proof*⟩

lemma *true-annot-insert-l*[*simp*]:
M $\models$*a A* $\Longrightarrow$ *L* # *M* $\models$*a A*
⟨*proof*⟩

lemma *true-annots-insert-l* [*simp*]:
M $\models$*as A* $\Longrightarrow$ *L* # *M* $\models$*as A*
⟨*proof*⟩

lemma *true-annots-union*[*iff*]:
M $\models$*as A* $\cup$ *B* $\longleftrightarrow$ (*M* $\models$*as A* $\land$ *M* $\models$*as B*)

$\langle proof \rangle$

lemma *true-annots-insert*[*iff*]:
$M \models as\ insert\ a\ A \longleftrightarrow (M \models a\ a \wedge M \models as\ A)$
$\langle proof \rangle$

Link between $\models as$ and $\models s$:

lemma *true-annots-true-cls*:
$I \models as\ CC \longleftrightarrow lits\text{-}of\text{-}l\ I \models s\ CC$
$\langle proof \rangle$

lemma *in-lit-of-true-annot*:
$a \in lits\text{-}of\text{-}l\ M \longleftrightarrow M \models a\ \{\#a\#\}$
$\langle proof \rangle$

lemma *true-annot-lit-of-notin-skip*:
$L \ \#\ M \models a\ A \Longrightarrow lit\text{-}of\ L \notin\#\ A \Longrightarrow M \models a\ A$
$\langle proof \rangle$

lemma *true-clss-singleton-lit-of-implies-incl*:
$I \models s\ unmark\text{-}l\ MLs \Longrightarrow lits\text{-}of\text{-}l\ MLs \subseteq I$
$\langle proof \rangle$

lemma *true-annot-true-clss-cls*:
$MLs \models a\ \psi \Longrightarrow set\ (map\ unmark\ MLs) \models p\ \psi$
$\langle proof \rangle$

lemma *true-annots-true-clss-cls*:
$MLs \models as\ \psi \Longrightarrow set\ (map\ unmark\ MLs) \models ps\ \psi$
$\langle proof \rangle$

lemma *true-annots-decided-true-cls*[*iff*]:
$map\ Decided\ M \models as\ N \longleftrightarrow set\ M \models s\ N$
$\langle proof \rangle$

lemma *true-annot-singleton*[*iff*]: $M \models a\ \{\#L\#\} \longleftrightarrow L \in lits\text{-}of\text{-}l\ M$
$\langle proof \rangle$

lemma *true-annots-true-clss-clss*:
$A \models as\ \Psi \Longrightarrow unmark\text{-}l\ A \models ps\ \Psi$
$\langle proof \rangle$

lemma *true-annot-commute*:
$M\ @\ M' \models a\ D \longleftrightarrow M'\ @\ M \models a\ D$
$\langle proof \rangle$

lemma *true-annots-commute*:
$M\ @\ M' \models as\ D \longleftrightarrow M'\ @\ M \models as\ D$
$\langle proof \rangle$

lemma *true-annot-mono*[*dest*]:
$set\ I \subseteq set\ I' \Longrightarrow I \models a\ N \Longrightarrow I' \models a\ N$
$\langle proof \rangle$

lemma *true-annots-mono*:

$set\ I \subseteq set\ I' \Longrightarrow I \models as\ N \Longrightarrow I' \models as\ N$
⟨*proof*⟩

Defined and undefined literals

We introduce the functions *defined-lit* and *undefined-lit* to know whether a literal is defined with respect to a list of decided literals (aka a trail in most cases).

Remark that *undefined* already exists and is a completely different Isabelle function.

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

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

lemma *defined-lit-rev*[*simp*]:
defined-lit (*rev* M) $L \longleftrightarrow$ *defined-lit* $M\ L$
⟨*proof*⟩

lemma *atm-imp-decided-or-proped*:
assumes $x \in$ *set* I
shows
(*Decided* ($-$ *lit-of* x) $\in$ *set* I)
$\vee$ (*Decided* (*lit-of* x) $\in$ *set* I)
$\vee$ ($\exists l.$ *Propagated* ($-$ *lit-of* x) $l \in$ *set* I)
$\vee$ ($\exists l.$ *Propagated* (*lit-of* x) $l \in$ *set* I)
⟨*proof*⟩

lemma *literal-is-lit-of-decided*:
assumes $L =$ *lit-of* x
shows ($x =$ *Decided* L) $\vee$ ($\exists l'.\ x =$ *Propagated* $L\ l'$)
⟨*proof*⟩

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

lemma *consistent-inter-true-annots-satisfiable*:
consistent-interp (*lits-of-l* I) $\Longrightarrow I \models as\ N \Longrightarrow$ *satisfiable* N
⟨*proof*⟩

lemma *defined-lit-map*:
defined-lit $Ls\ L \longleftrightarrow$ *atm-of* $L \in (\lambda l.$ *atm-of* (*lit-of* l)) ` *set* Ls
⟨*proof*⟩

lemma *defined-lit-uminus*[*iff*]:
defined-lit $I\ (-L) \longleftrightarrow$ *defined-lit* $I\ L$
⟨*proof*⟩

lemma *Decided-Propagated-in-iff-in-lits-of-l*:
defined-lit $I\ L \longleftrightarrow$ ($L \in$ *lits-of-l* $I \vee -L \in$ *lits-of-l* I)
⟨*proof*⟩

lemma *consistent-add-undefined-lit-consistent*[*simp*]:

assumes
  consistent-interp (*lits-of-l Ls*) **and**
  undefined-lit Ls L
shows *consistent-interp* (*insert L* (*lits-of-l Ls*))
⟨*proof*⟩

lemma *decided-empty*[*simp*]:
  ¬*defined-lit* [] *L*
  ⟨*proof*⟩

4.3.2 Backtracking

fun *backtrack-split* :: ('*v*, '*m*) *ann-lits*
  ⇒ ('*v*, '*m*) *ann-lits* × ('*v*, '*m*) *ann-lits* **where**
backtrack-split [] = ([], []) |
backtrack-split (*Propagated L P* # *mlits*) = *apfst* ((*op* #) (*Propagated L P*)) (*backtrack-split mlits*) |
backtrack-split (*Decided L* # *mlits*) = ([], *Decided L* # *mlits*)

lemma *backtrack-split-fst-not-decided*: $a \in$ *set* (*fst* (*backtrack-split l*)) ⟹ ¬*is-decided a*
  ⟨*proof*⟩

lemma *backtrack-split-snd-hd-decided*:
  snd (*backtrack-split l*) ≠ [] ⟹ *is-decided* (*hd* (*snd* (*backtrack-split l*)))
  ⟨*proof*⟩

lemma *backtrack-split-list-eq*[*simp*]:
  fst (*backtrack-split l*) @ (*snd* (*backtrack-split l*)) = *l*
  ⟨*proof*⟩

lemma *backtrack-snd-empty-not-decided*:
  backtrack-split M = (M'', []) ⟹ ∀ *l*∈*set M*. ¬ *is-decided l*
  ⟨*proof*⟩

lemma *backtrack-split-some-is-decided-then-snd-has-hd*:
  ∃ *l*∈*set M*. *is-decided l* ⟹ ∃ M' L' M''. *backtrack-split M* = (M'', L' # M')
  ⟨*proof*⟩

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:
  backtrack-split M = (*takeWhile* (*Not o is-decided*) *M*, *dropWhile* (*Not o is-decided*) *M*)
  ⟨*proof*⟩

4.3.3 Decomposition with respect to the First Decided Literals

In this section we define a function that returns a decomposition with the first decided literal. This function is useful to define the backtracking of DPLL.

Definition

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

fun *get-all-ann-decomposition* :: ('*a*, '*m*) *ann-lits*
  ⇒ (('*a*, '*m*) *ann-lits* × ('*a*, '*m*) *ann-lits*) *list* **where**

get-all-ann-decomposition (*Decided L* # *Ls*) =
(*Decided L* # *Ls*, []) # *get-all-ann-decomposition Ls* |
get-all-ann-decomposition (*Propagated L P*# *Ls*) =
(*apsnd* ((*op* #) (*Propagated L P*)) (*hd* (*get-all-ann-decomposition Ls*)))
tl (*get-all-ann-decomposition Ls*) |
get-all-ann-decomposition [] = [([], [])]

value *get-all-ann-decomposition* [*Propagated A5 B5*, *Decided C4*, *Propagated A3 B3*, *Propagated A2 B2*, *Decided C1*, *Propagated A0 B0*]

Now we can prove several simple properties about the function.

lemma *get-all-ann-decomposition-never-empty*[*iff*]:
get-all-ann-decomposition M = [] $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *get-all-ann-decomposition-never-empty-sym*[*iff*]:
[] = *get-all-ann-decomposition M* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *get-all-ann-decomposition-decomp*:
hd (*get-all-ann-decomposition S*) = (*a*, *c*) $\Longrightarrow$ *S* = *c* @ *a*
⟨*proof*⟩

lemma *get-all-ann-decomposition-backtrack-split*:
backtrack-split S = (*M*, *M'*) $\longleftrightarrow$ *hd* (*get-all-ann-decomposition S*) = (*M'*, *M*)
⟨*proof*⟩

lemma *get-all-ann-decomposition-Nil-backtrack-split-snd-Nil*:
get-all-ann-decomposition S = [([], *A*)] $\Longrightarrow$ *snd* (*backtrack-split S*) = []
⟨*proof*⟩

This functions says that the first element is either empty or starts with a decided element of the list.

lemma *get-all-ann-decomposition-length-1-fst-empty-or-length-1*:
assumes *get-all-ann-decomposition M* = (*a*, *b*) # []
shows *a* = [] $\vee$ (*length a* = *1* $\wedge$ *is-decided* (*hd a*) $\wedge$ *hd a* $\in$ *set M*)
⟨*proof*⟩

lemma *get-all-ann-decomposition-fst-empty-or-hd-in-M*:
assumes *get-all-ann-decomposition M* = (*a*, *b*) # *l*
shows *a* = [] $\vee$ (*is-decided* (*hd a*) $\wedge$ *hd a* $\in$ *set M*)
⟨*proof*⟩

lemma *get-all-ann-decomposition-snd-not-decided*:
assumes (*a*, *b*) $\in$ *set* (*get-all-ann-decomposition M*)
and *L* $\in$ *set b*
shows $\neg$*is-decided L*
⟨*proof*⟩

lemma *tl-get-all-ann-decomposition-skip-some*:
assumes *x* $\in$ *set* (*tl* (*get-all-ann-decomposition M1*))
shows *x* $\in$ *set* (*tl* (*get-all-ann-decomposition* (*M0* @ *M1*)))
⟨*proof*⟩

lemma *hd-get-all-ann-decomposition-skip-some*:

assumes $(x,\ y) = hd\ (get\text{-}all\text{-}ann\text{-}decomposition\ M1)$
shows $(x,\ y) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M0\ @\ Decided\ K\ \#\ M1))$
$\langle proof \rangle$

lemma *in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend*:
$(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M') \Longrightarrow$
$\exists\, b'.\ (a,\ b'\ @\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M\ @\ M'))$
$\langle proof \rangle$

lemma *in-get-all-ann-decomposition-decided-or-empty*:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $a = [] \vee (is\text{-}decided\ (hd\ a))$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-remove-undecided-length*:
assumes $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
shows $length\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ M'')) = length\ (get\text{-}all\text{-}ann\text{-}decomposition\ M'')$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-not-is-decided-length*:
assumes $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
shows $1 + length\ (get\text{-}all\text{-}ann\text{-}decomposition\ (Propagated\ (-L)\ P\ \#\ M))$
$= length\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ Decided\ L\ \#\ M))$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-last-choice*:
assumes $tl\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ Decided\ L\ \#\ M)) \neq []$
and $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
and $hd\ (tl\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ Decided\ L\ \#\ M))) = (M0',\ M0)$
shows $hd\ (get\text{-}all\text{-}ann\text{-}decomposition\ (Propagated\ (-L)\ P\ \#\ M)) = (M0',\ Propagated\ (-L)\ P\ \#\ M0)$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-except-last-choice-equal*:
assumes $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
shows $tl\ (get\text{-}all\text{-}ann\text{-}decomposition\ (Propagated\ (-L)\ P\ \#\ M))$
$= tl\ (tl\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ Decided\ L\ \#\ M)))$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-hd-hd*:
assumes $get\text{-}all\text{-}ann\text{-}decomposition\ Ls = (M,\ C)\ \#\ (M0,\ M0')\ \#\ l$
shows $tl\ M = M0'\ @\ M0 \wedge is\text{-}decided\ (hd\ M)$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-exists-prepend*[*dest*]:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $\exists\, c.\ M = c\ @\ b\ @\ a$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-incl*:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $set\ b \subseteq set\ M$ **and** $set\ a \subseteq set\ M$
$\langle proof \rangle$

lemma *get-all-ann-decomposition-exists-prepend'*:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
obtains c **where** $M = c\ @\ b\ @\ a$

$\langle proof \rangle$

lemma *union-in-get-all-ann-decomposition-is-subset*:
assumes $(a, b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $set\ a \cup set\ b \subseteq set\ M$
$\langle proof \rangle$

lemma *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons*:
$\exists\, M1\ M2.\ (Decided\ K\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (c\ @\ Decided\ K\ \#\ c'))$
$\langle proof \rangle$

lemma *fst-get-all-ann-decomposition-prepend-not-decided*:
assumes $\forall\, m{\in}set\ MS.\ \neg\ is\text{-}decided\ m$
shows $set\ (map\ fst\ (get\text{-}all\text{-}ann\text{-}decomposition\ M))$
$= set\ (map\ fst\ (get\text{-}all\text{-}ann\text{-}decomposition\ (MS\ @\ M)))$
$\langle proof \rangle$

Entailment of the Propagated by the Decided Literal

lemma *get-all-ann-decomposition-snd-union*:
$set\ M = \bigcup(set\ `\ snd\ `\ set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)) \cup \{L\ |L.\ is\text{-}decided\ L \wedge L \in set\ M\}$
(**is** $?M\ M = ?U\ M \cup ?Ls\ M$)
$\langle proof \rangle$

definition *all-decomposition-implies* :: $'a\ literal\ multiset\ set$
$\Rightarrow (('a, 'm)\ ann\text{-}lits \times ('a, 'm)\ ann\text{-}lits)\ list \Rightarrow bool$ **where**
$all\text{-}decomposition\text{-}implies\ N\ S \longleftrightarrow (\forall\,(Ls, seen) \in set\ S.\ unmark\text{-}l\ Ls \cup N \models ps\ unmark\text{-}l\ seen)$

lemma *all-decomposition-implies-empty*[*iff*]:
$all\text{-}decomposition\text{-}implies\ N\ []$ $\langle proof \rangle$

lemma *all-decomposition-implies-single*[*iff*]:
$all\text{-}decomposition\text{-}implies\ N\ [(Ls, seen)] \longleftrightarrow unmark\text{-}l\ Ls \cup N \models ps\ unmark\text{-}l\ seen$
$\langle proof \rangle$

lemma *all-decomposition-implies-append*[*iff*]:
$all\text{-}decomposition\text{-}implies\ N\ (S\ @\ S')$
$\longleftrightarrow (all\text{-}decomposition\text{-}implies\ N\ S \wedge all\text{-}decomposition\text{-}implies\ N\ S')$
$\langle proof \rangle$

lemma *all-decomposition-implies-cons-pair*[*iff*]:
$all\text{-}decomposition\text{-}implies\ N\ ((Ls, seen)\ \#\ S')$
$\longleftrightarrow (all\text{-}decomposition\text{-}implies\ N\ [(Ls, seen)] \wedge all\text{-}decomposition\text{-}implies\ N\ S')$
$\langle proof \rangle$

lemma *all-decomposition-implies-cons-single*[*iff*]:
$all\text{-}decomposition\text{-}implies\ N\ (l\ \#\ S') \longleftrightarrow$
$(unmark\text{-}l\ (fst\ l) \cup N \models ps\ unmark\text{-}l\ (snd\ l) \wedge$
$all\text{-}decomposition\text{-}implies\ N\ S')$
$\langle proof \rangle$

lemma *all-decomposition-implies-trail-is-implied*:
assumes $all\text{-}decomposition\text{-}implies\ N\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $N \cup \{unmark\ L\ |L.\ is\text{-}decided\ L \wedge L \in set\ M\}$
$\models ps\ unmark\ `\ \bigcup(set\ `\ snd\ `\ set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M))$
$\langle proof \rangle$

lemma *all-decomposition-implies-propagated-lits-are-implied*:
  assumes *all-decomposition-implies N* (*get-all-ann-decomposition M*)
  shows $N \cup \{unmark\ L\ |L.\ is\text{-}decided\ L \wedge L \in set\ M\} \models ps\ unmark\text{-}l\ M$
    (**is** $?I \models ps\ ?A$)
⟨*proof*⟩

lemma *all-decomposition-implies-insert-single*:
  $all\text{-}decomposition\text{-}implies\ N\ M \Longrightarrow all\text{-}decomposition\text{-}implies\ (insert\ C\ N)\ M$
  ⟨*proof*⟩

4.3.4 Negation of Clauses

We define the negation of a *'a Partial-Clausal-Logic.clause*: it converts it from the a single clause to a set of clauses, wherein each clause is a single negated literal.

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

lemma *in-CNot-uminus*[*iff*]:
  shows $\{\#L\#\} \in CNot\ \psi \longleftrightarrow -L \in\#\ \psi$
  ⟨*proof*⟩

lemma
  shows
    CNot-singleton[*simp*]: $CNot\ \{\#L\#\} = \{\{\#-L\#\}\}$ **and**
    CNot-empty[*simp*]: $CNot\ \{\#\} = \{\}$ **and**
    CNot-plus[*simp*]: $CNot\ (A + B) = CNot\ A \cup CNot\ B$
  ⟨*proof*⟩

lemma *CNot-eq-empty*[*iff*]:
  $CNot\ D = \{\} \longleftrightarrow D = \{\#\}$
  ⟨*proof*⟩

lemma *in-CNot-implies-uminus*:
  assumes $L \in\#\ D$ **and** $M \models as\ CNot\ D$
  shows $M \models a\ \{\#-L\#\}$ **and** $-L \in lits\text{-}of\text{-}l\ M$
  ⟨*proof*⟩

lemma *CNot-remdups-mset*[*simp*]:
  $CNot\ (remdups\text{-}mset\ A) = CNot\ A$
  ⟨*proof*⟩

lemma *Ball-CNot-Ball-mset*[*simp*]:
  $(\forall x \in CNot\ D.\ P\ x) \longleftrightarrow (\forall L \in\#\ D.\ P\ \{\#-L\#\})$
 ⟨*proof*⟩

lemma *consistent-CNot-not*:
  assumes *consistent-interp I*
  shows $I \models s\ CNot\ \varphi \Longrightarrow \neg I \models \varphi$
  ⟨*proof*⟩

lemma *total-not-true-cls-true-clss-CNot*:
  assumes $total\text{-}over\text{-}m\ I\ \{\varphi\}$ **and** $\neg I \models \varphi$
  shows $I \models s\ CNot\ \varphi$
  ⟨*proof*⟩

lemma *total-not-CNot*:
assumes *total-over-m* I $\{\varphi\}$ **and** $\neg I \models s$ *CNot* φ
shows $I \models \varphi$
⟨*proof*⟩

lemma *atms-of-ms-CNot-atms-of*[*simp*]:
atms-of-ms (*CNot* C) = *atms-of* C
⟨*proof*⟩

lemma *true-clss-clss-contradiction-true-clss-cls-false*:
$C \in D \Longrightarrow D \models ps$ *CNot* $C \Longrightarrow D \models p$ $\{\#\}$
⟨*proof*⟩

lemma *true-annots-CNot-all-atms-defined*:
assumes $M \models as$ *CNot* T **and** *a1*: $L \in\#$ T
shows *atm-of* $L \in$ *atm-of* ' *lits-of-l* M
⟨*proof*⟩

lemma *true-annots-CNot-all-uminus-atms-defined*:
assumes $M \models as$ *CNot* T **and** *a1*: $-L \in\#$ T
shows *atm-of* $L \in$ *atm-of* ' *lits-of-l* M
⟨*proof*⟩

lemma *true-clss-clss-false-left-right*:
assumes $\{\{\#L\#\}\} \cup B \models p$ $\{\#\}$
shows $B \models ps$ *CNot* $\{\#L\#\}$
⟨*proof*⟩

lemma *true-annots-true-cls-def-iff-negation-in-model*:
$M \models as$ *CNot* $C \longleftrightarrow (\forall L \in\# C. -L \in$ *lits-of-l* $M)$
⟨*proof*⟩

lemma *true-annot-CNot-diff*:
$I \models as$ *CNot* $C \Longrightarrow I \models as$ *CNot* $(C - C')$
⟨*proof*⟩

lemma *CNot-mset-replicate*[*simp*]:
CNot (*mset* (*replicate* n L)) = (*if* $n = 0$ *then* {} *else* $\{\{\#-L\#\}\}$)
⟨*proof*⟩

lemma *consistent-CNot-not-tautology*:
consistent-interp $M \Longrightarrow M \models s$ *CNot* $D \Longrightarrow \neg$*tautology* D
⟨*proof*⟩

lemma *atms-of-ms-CNot-atms-of-ms*: *atms-of-ms* (*CNot* CC) = *atms-of-ms* $\{CC\}$
⟨*proof*⟩

lemma *total-over-m-CNot-toal-over-m*[*simp*]:
total-over-m I (*CNot* C) = *total-over-set* I (*atms-of* C)
⟨*proof*⟩

The following lemma is very useful when in the goal appears an axioms like $-L = K$: this lemma allows the simplifier to rewrite L.

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

$\langle proof \rangle$

lemma *true-clss-cls-plus-CNot*:
assumes
CC-L: $A \models p\ CC + \{\#L\#\}$ **and**
CNot-CC: $A \models ps\ CNot\ CC$
shows $A \models p\ \{\#L\#\}$
$\langle proof \rangle$

lemma *true-annots-CNot-lit-of-notin-skip*:
assumes *LM*: $L\ \#\ M \models as\ CNot\ A$ **and** *LA*: $lit\text{-}of\ L \notin\#\ A\ {-lit\text{-}of}\ L \notin\#\ A$
shows $M \models as\ CNot\ A$
$\langle proof \rangle$

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:
$A \cup \{B\} \models ps\ \{\{\#\}\} \longleftrightarrow A \models ps\ CNot\ B$
$\langle proof \rangle$

lemma *true-annot-remove-hd-if-notin-vars*:
assumes $a\ \#\ M' \models a\ D$ **and** $atm\text{-}of\ (lit\text{-}of\ a) \notin atms\text{-}of\ D$
shows $M' \models a\ D$
$\langle proof \rangle$

lemma *true-annot-remove-if-notin-vars*:
assumes $M\ @\ M' \models a\ D$ **and** $\forall x{\in}atms\text{-}of\ D.\ x \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $M' \models a\ D$
$\langle proof \rangle$

lemma *true-annots-remove-if-notin-vars*:
assumes $M\ @\ M' \models as\ D$ **and** $\forall x{\in}atms\text{-}of\text{-}ms\ D.\ x \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $M' \models as\ D$ $\langle proof \rangle$

lemma *all-variables-defined-not-imply-cnot*:
assumes
$\forall s \in atms\text{-}of\text{-}ms\ \{B\}.\ s \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ A$ **and**
$\neg\ A \models a\ B$
shows $A \models as\ CNot\ B$
$\langle proof \rangle$

lemma *CNot-union-mset*[*simp*]:
$CNot\ (A\ \#{\cup}\ B) = CNot\ A \cup CNot\ B$
$\langle proof \rangle$

4.3.5 Other

abbreviation $no\text{-}dup\ L \equiv distinct\ (map\ (\lambda l.\ atm\text{-}of\ (lit\text{-}of\ l))\ L)$

lemma *no-dup-rev*[*simp*]:
$no\text{-}dup\ (rev\ M) \longleftrightarrow no\text{-}dup\ M$
$\langle proof \rangle$

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
assumes $no\text{-}dup\ M$
shows $length\ M = card\ (atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M)$
$\langle proof \rangle$

lemma *distinct-consistent-interp*:
no-dup M $\Longrightarrow$ *consistent-interp* (*lits-of-l M*)
⟨*proof*⟩

lemma *distinct-get-all-ann-decomposition-no-dup*:
assumes $(a, b) \in$ *set* (*get-all-ann-decomposition M*)
and *no-dup M*
shows *no-dup* (a @ b)
⟨*proof*⟩

lemma *true-annots-lit-of-notin-skip*:
assumes L # M $\models$*as CNot A*
and $-$*lit-of* $L \notin\#$ A
and *no-dup* (L # M)
shows $M \models$*as CNot A*
⟨*proof*⟩

4.3.6 Extending Entailments to multisets

We have defined previous entailment with respect to sets, but we also need a multiset version depending on the context. The conversion is simple using the function *set-mset* (in this direction, there is no loss of information).

abbreviation *true-annots-mset* (**infix** $\models$*asm 50*) **where**
$I \models$*asm* $C \equiv I \models$*as* (*set-mset* C)

abbreviation *true-clss-clss-m*:: *'v clause multiset* $\Rightarrow$ *'v clause multiset* $\Rightarrow$ *bool* (**infix** $\models$*psm 50*)
where
$I \models$*psm* $C \equiv$ *set-mset* $I \models$*ps* (*set-mset* C)

Analog of theorem *true-clss-clss-subsetE*

lemma *true-clss-clssm-subsetE*: $N \models$*psm* $B \Longrightarrow A \subseteq\# B \Longrightarrow N \models$*psm* A
⟨*proof*⟩

abbreviation *true-clss-cls-m*:: *'a clause multiset* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool* (**infix** $\models$*pm 50*) **where**
$I \models$*pm* $C \equiv$ *set-mset* $I \models$*p* C

abbreviation *distinct-mset-mset* :: *'a multiset multiset* $\Rightarrow$ *bool* **where**
distinct-mset-mset $\Sigma \equiv$ *distinct-mset-set* (*set-mset* Σ)

abbreviation *all-decomposition-implies-m* **where**
all-decomposition-implies-m A $B \equiv$ *all-decomposition-implies* (*set-mset* A) B

abbreviation *atms-of-mm* :: *'a literal multiset multiset* $\Rightarrow$ *'a set* **where**
atms-of-mm $U \equiv$ *atms-of-ms* (*set-mset* U)

Other definition using *Union-mset*

lemma *atms-of-mm* $U \equiv$ *set-mset* ($\bigcup\#$ *image-mset* (*image-mset atm-of*) U)
⟨*proof*⟩

abbreviation *true-clss-m*:: *'a interp* $\Rightarrow$ *'a clause multiset* $\Rightarrow$ *bool* (**infix** $\models$*sm 50*) **where**
$I \models$*sm* $C \equiv I \models$*s set-mset* C

abbreviation *true-clss-ext-m* (**infix** $\models$*sextm 49*) **where**
$I \models$*sextm* $C \equiv I \models$*sext set-mset* C

type-synonym $'v$ *clauses* $=$ $'v$ *clause multiset*
end

Chapter 5

NOT's CDCL and DPLL

theory *CDCL-WNOT-Measure*
imports *Main List-More*
begin

The organisation of the development is the following:

- `CDCL_WNOT_Measure.thy` contains the measure used to show the termination the core of CDCL.
- `CDCL_NOT.thy` contains the specification of the rules: the rules are defined, and we proof the correctness and termination for some strategies CDCL.
- `DPLL_NOT.thy` contains the DPLL calculus based on the CDCL version.
- `DPLL_W.thy` contains Weidenbach's version of DPLL and the proof of equivalence between the two DPLL versions.

5.1 Measure

This measure show the termination of the core of CDCL: each step improves the number of literals we know for sure.

This measure can also be seen as the increasing lexicographic order: it is an order on bounded sequences, when each element is bounded. The proof involves a measure like the one defined here (the same?).

definition μ_C :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *nat* **where**
$\mu_C\ s\ b\ M \equiv (\sum i{=}0..{<}length\ M.\ M!i * b\hat{}\ (s + i - length\ M))$

lemma μ_C-*Nil*[*simp*]:
  $\mu_C\ s\ b\ [] = 0$
  ⟨*proof*⟩

lemma μ_C-*single*[*simp*]:
  $\mu_C\ s\ b\ [L] = L * b\ \hat{}\ (s - Suc\ 0)$
  ⟨*proof*⟩

lemma *set-sum-atLeastLessThan-add*:
  $(\sum i{=}k..{<}k{+}(b{::}nat).\ f\ i) = (\sum i{=}0..{<}b.\ f\ (k{+}\ i))$
  ⟨*proof*⟩

lemma *set-sum-atLeastLessThan-Suc*:
  $(\sum i{=}1..{<}Suc\ j.\ f\ i) = (\sum i{=}0..{<}j.\ f\ (Suc\ i))$
  ⟨*proof*⟩

lemma μ_C*-cons*:
  $\mu_C\ s\ b\ (L\ \#\ M) = L * b\ \hat{}\ (s - 1 - length\ M) + \mu_C\ s\ b\ M$
⟨*proof*⟩

lemma μ_C*-append*:
  assumes $s \geq length\ (M@M')$
  shows $\mu_C\ s\ b\ (M@M') = \mu_C\ (s - length\ M')\ b\ M + \mu_C\ s\ b\ M'$
⟨*proof*⟩

lemma μ_C*-cons-non-empty-inf*:
  assumes *M-ge-1*: $\forall\, i{\in}set\ M.\ i \geq 1$ **and** *M*: $M \neq []$
  shows $\mu_C\ s\ b\ M \geq b\ \hat{}\ (s - length\ M)$
  ⟨*proof*⟩

Copy of `~~/src/HOL/ex/NatSum.thy` (but generalized to $0 \leq k$)

lemma *sum-of-powers*: $0 \leq k \Longrightarrow (k - 1) * (\sum i{=}0..{<}n.\ k\hat{}i) = k\hat{}n - (1::nat)$
  ⟨*proof*⟩

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

lemma μ_C*-bounded-non-degenerated*:
  fixes b ::*nat*
  assumes
    $b > 0$ **and**
    $M \neq []$ **and**
    M-le: $\forall\, i < length\ M.\ M!i < b$ **and**
    $s \geq length\ M$
  shows $\mu_C\ s\ b\ M < b\hat{}s$
⟨*proof*⟩

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

lemma μ_C*-bounded*:
  fixes b :: *nat*
  assumes
    M-le: $\forall\, i < length\ M.\ M!i < b$ **and**
    $s \geq length\ M$
    $b > 0$
  shows $\mu_C\ s\ b\ M < b\ \hat{}\ s$
⟨*proof*⟩

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

lemma μ_C*-base-0*:
  assumes $length\ M \leq s$
  shows $\mu_C\ s\ 0\ M \leq M!0$
⟨*proof*⟩

lemma *finite-bounded-pair-list*:
  fixes b :: *nat*
  shows *finite* $\{(ys,\ xs).\ length\ xs < s \land length\ ys < s \land$

$(\forall\, i< \textit{length xs. xs} ! \ i < b) \wedge (\forall\, i< \textit{length ys. ys} ! \ i < b)\}$
⟨*proof*⟩

definition νNOT :: *nat* ⇒ *nat* ⇒ (*nat list* × *nat list*) *set* **where**
νNOT *s base* = {(*ys*, *xs*). *length xs* < *s* ∧ *length ys* < *s* ∧
(∀ *i*< *length xs*. *xs* ! *i* < *base*) ∧ (∀ *i*< *length ys*. *ys* ! *i* < *base*) ∧
(*ys*, *xs*) ∈ *lenlex less-than*}

lemma *finite-*νNOT[*simp*]:
finite (νNOT *s base*)
⟨*proof*⟩

lemma *acyclic-*νNOT: *acyclic* (νNOT *s base*)
⟨*proof*⟩

lemma *wf-*νNOT: *wf* (νNOT *s base*)
⟨*proof*⟩

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

5.2 NOT's CDCL

5.2.1 Auxiliary Lemmas and Measure

We define here some more simplification rules, or rules that have been useful as help for some tactic

lemma *no-dup-cannot-not-lit-and-uminus*:
no-dup M ⟹ − *lit-of xa* = *lit-of x* ⟹ *x* ∈ *set M* ⟹ *xa* ∉ *set M*
⟨*proof*⟩

lemma *atms-of-ms-single-atm-of*[*simp*]:
atms-of-ms {*unmark L* |*L*. *P L*} = *atm-of* ‘ {*lit-of L* |*L*. *P L*}
⟨*proof*⟩

lemma *atms-of-uminus-lit-atm-of-lit-of*:
atms-of {# −*lit-of x*. *x* ∈# *A*#} = *atm-of* ‘ (*lit-of* ‘ (*set-mset A*))
⟨*proof*⟩

lemma *atms-of-ms-single-image-atm-of-lit-of*:
atms-of-ms (*unmark-s A*) = *atm-of* ‘ (*lit-of* ‘ *A*)
⟨*proof*⟩

5.2.2 Initial definitions

The state

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

locale *dpll-state-ops* =
fixes
trail :: ′*st* ⇒ (′*v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: ′*st* ⇒ ′*v clauses* **and**

prepend-trail :: ('*v*, *unit*) *ann-lit* ⇒ '*st* ⇒ '*st* **and**
tl-trail :: '*st* ⇒'*st* **and**
$add\text{-}cls_{NOT}$ :: '*v clause* ⇒ '*st* ⇒ '*st* **and**
$remove\text{-}cls_{NOT}$ :: '*v clause* ⇒ '*st* ⇒ '*st*
begin
abbreviation $state_{NOT}$ :: '*st* ⇒ ('*v*, *unit*) *ann-lit list* × '*v clauses* **where**
$state_{NOT}\ S \equiv (trail\ S,\ clauses_{NOT}\ S)$
end

NOT's state is basically a pair composed of the trail (i.e. the candidate model) and the set of clauses. We abstract this state to convert this state to other states. like Weidenbach's five-tuple.

locale *dpll-state* =
dpll-state-ops
trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ — related to the state
for
trail :: '*st* ⇒ ('*v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: '*st* ⇒ '*v clauses* **and**
prepend-trail :: ('*v*, *unit*) *ann-lit* ⇒ '*st* ⇒ '*st* **and**
tl-trail :: '*st* ⇒'*st* **and**
$add\text{-}cls_{NOT}$ :: '*v clause* ⇒ '*st* ⇒ '*st* **and**
$remove\text{-}cls_{NOT}$ :: '*v clause* ⇒ '*st* ⇒ '*st* +
assumes
$prepend\text{-}trail_{NOT}$:
$state_{NOT}$ (*prepend-trail L st*) = (*L* # *trail st*, $clauses_{NOT}$ *st*) **and**
$tl\text{-}trail_{NOT}$:
$state_{NOT}$ (*tl-trail st*) = (*tl* (*trail st*), $clauses_{NOT}$ *st*) **and**
$add\text{-}cls_{NOT}$:
$state_{NOT}$ ($add\text{-}cls_{NOT}$ *C st*) = (*trail st*, {#*C*#} + $clauses_{NOT}$ *st*) **and**
$remove\text{-}cls_{NOT}$:
$state_{NOT}$ ($remove\text{-}cls_{NOT}$ *C st*) = (*trail st*, *removeAll-mset C* ($clauses_{NOT}$ *st*))
begin
lemma
trail-prepend-trail[*simp*]:
trail (*prepend-trail L st*) = *L* # *trail st*
and
$trail\text{-}tl\text{-}trail_{NOT}$[*simp*]: *trail* (*tl-trail st*) = *tl* (*trail st*) **and**
$trail\text{-}add\text{-}cls_{NOT}$[*simp*]: *trail* ($add\text{-}cls_{NOT}$ *C st*) = *trail st* **and**
$trail\text{-}remove\text{-}cls_{NOT}$[*simp*]: *trail* ($remove\text{-}cls_{NOT}$ *C st*) = *trail st* **and**

clauses-prepend-trail[*simp*]:
$clauses_{NOT}$ (*prepend-trail L st*) = $clauses_{NOT}$ *st*
and
clauses-tl-trail[*simp*]: $clauses_{NOT}$ (*tl-trail st*) = $clauses_{NOT}$ *st* **and**
$clauses\text{-}add\text{-}cls_{NOT}$[*simp*]:
$clauses_{NOT}$ ($add\text{-}cls_{NOT}$ *C st*) = {#*C*#} + $clauses_{NOT}$ *st* **and**
$clauses\text{-}remove\text{-}cls_{NOT}$[*simp*]:
$clauses_{NOT}$ ($remove\text{-}cls_{NOT}$ *C st*) = *removeAll-mset C* ($clauses_{NOT}$ *st*)
⟨*proof*⟩

We define the following function doing the backtrack in the trail:

function $reduce\text{-}trail\text{-}to_{NOT}$ :: '*a list* ⇒ '*st* ⇒ '*st* **where**
$reduce\text{-}trail\text{-}to_{NOT}$ *F S* =
(*if length* (*trail S*) = *length F* ∨ *trail S* = [] *then S else* $reduce\text{-}trail\text{-}to_{NOT}$ *F* (*tl-trail S*))
⟨*proof*⟩
termination ⟨*proof*⟩

declare *reduce-trail-to$_{NOT}$.simps*[*simp del*]

Then we need several lemmas about the *reduce-trail-to$_{NOT}$*.

lemma
  shows
  reduce-trail-to$_{NOT}$-Nil[*simp*]: *trail S* = [] $\Longrightarrow$ *reduce-trail-to$_{NOT}$ F S* = *S* **and**
  reduce-trail-to$_{NOT}$-eq-length[*simp*]: *length* (*trail S*) = *length F* $\Longrightarrow$ *reduce-trail-to$_{NOT}$ F S* = *S*
  ⟨*proof*⟩

lemma *reduce-trail-to$_{NOT}$-length-ne*[*simp*]:
  length (*trail S*) $\neq$ *length F* $\Longrightarrow$ *trail S* $\neq$ [] $\Longrightarrow$
    reduce-trail-to$_{NOT}$ F S = *reduce-trail-to$_{NOT}$ F* (*tl-trail S*)
  ⟨*proof*⟩

lemma *trail-reduce-trail-to$_{NOT}$-length-le*:
  assumes *length F* > *length* (*trail S*)
  shows *trail* (*reduce-trail-to$_{NOT}$ F S*) = []
  ⟨*proof*⟩

lemma *trail-reduce-trail-to$_{NOT}$-Nil*[*simp*]:
  trail (*reduce-trail-to$_{NOT}$* [] *S*) = []
  ⟨*proof*⟩

lemma *clauses-reduce-trail-to$_{NOT}$-Nil*:
  clauses$_{NOT}$ (*reduce-trail-to$_{NOT}$* [] *S*) = *clauses$_{NOT}$ S*
  ⟨*proof*⟩

lemma *trail-reduce-trail-to$_{NOT}$-drop*:
  trail (*reduce-trail-to$_{NOT}$ F S*) =
    (*if length* (*trail S*) $\geq$ *length F*
    then drop (*length* (*trail S*) $-$ *length F*) (*trail S*)
    else [])
  ⟨*proof*⟩

lemma *reduce-trail-to$_{NOT}$-skip-beginning*:
  assumes *trail S* = *F'* @ *F*
  shows *trail* (*reduce-trail-to$_{NOT}$ F S*) = *F*
  ⟨*proof*⟩

lemma *reduce-trail-to$_{NOT}$-clauses*[*simp*]:
  clauses$_{NOT}$ (*reduce-trail-to$_{NOT}$ F S*) = *clauses$_{NOT}$ S*
  ⟨*proof*⟩

lemma *trail-eq-reduce-trail-to$_{NOT}$-eq*:
  trail S = *trail T* $\Longrightarrow$ *trail* (*reduce-trail-to$_{NOT}$ F S*) = *trail* (*reduce-trail-to$_{NOT}$ F T*)
  ⟨*proof*⟩

lemma *trail-reduce-trail-to$_{NOT}$-add-cls$_{NOT}$*[*simp*]:
  no-dup (*trail S*) $\Longrightarrow$
    trail (*reduce-trail-to$_{NOT}$ F* (*add-cls$_{NOT}$ C S*)) = *trail* (*reduce-trail-to$_{NOT}$ F S*)
  ⟨*proof*⟩

lemma *reduce-trail-to$_{NOT}$-trail-tl-trail-decomp*[*simp*]:
  trail S = *F'* @ *Decided K* # *F* $\Longrightarrow$
    trail (*reduce-trail-to$_{NOT}$ F* (*tl-trail S*)) = *F*
  ⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-length*:
  length M = *length M'* $\Longrightarrow$ *reduce-trail-to*$_{NOT}$ *M S* = *reduce-trail-to*$_{NOT}$ *M' S*
  ⟨*proof*⟩

abbreviation *trail-weight* **where**
trail-weight S ≡ *map* ((λ*l*. *1* + *length l*) *o snd*) (*get-all-ann-decomposition* (*trail S*))

As we are defining abstract states, the Isabelle equality about them is too strong: we want the weaker equivalence stating that two states are equal if they cannot be distinguished, i.e. given the getter *trail* and *clauses*$_{NOT}$ do not distinguish them.

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

lemma *state-eq*$_{NOT}$*-ref*[*simp*]:
  S ∼ *S*
  ⟨*proof*⟩

lemma *state-eq*$_{NOT}$*-sym*:
  S ∼ *T* ⟷ *T* ∼ *S*
  ⟨*proof*⟩

lemma *state-eq*$_{NOT}$*-trans*:
  S ∼ *T* $\Longrightarrow$ *T* ∼ *U* $\Longrightarrow$ *S* ∼ *U*
  ⟨*proof*⟩

lemma
  shows
    state-eq$_{NOT}$*-trail*: *S* ∼ *T* $\Longrightarrow$ *trail S* = *trail T* **and**
    state-eq$_{NOT}$*-clauses*: *S* ∼ *T* $\Longrightarrow$ *clauses*$_{NOT}$ *S* = *clauses*$_{NOT}$ *T*
  ⟨*proof*⟩

lemmas *state-simp*$_{NOT}$[*simp*] = *state-eq*$_{NOT}$*-trail state-eq*$_{NOT}$*-clauses*

lemma *reduce-trail-to*$_{NOT}$*-state-eq*$_{NOT}$*-compatible*:
  assumes *ST*: *S* ∼ *T*
  shows *reduce-trail-to*$_{NOT}$ *F S* ∼ *reduce-trail-to*$_{NOT}$ *F T*
⟨*proof*⟩

end

Definition of the operation

Each possible is in its own locale.

locale *propagate-ops* =
  dpll-state trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
  for
    trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
    clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
    prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
    tl-trail :: *'st* ⇒*'st* **and**
    add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
    remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* +
  fixes
    propagate-cond :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool*

begin
inductive $propagate_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$propagate_{NOT}[intro]$: $C + \{\#L\#\} \in\# \ clauses_{NOT}\ S \Longrightarrow trail\ S \models as\ CNot\ C$
$\Longrightarrow undefined\text{-}lit\ (trail\ S)\ L$
$\Longrightarrow propagate\text{-}cond\ (Propagated\ L\ ())\ S$
$\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ S$
$\Longrightarrow propagate_{NOT}\ S\ T$
inductive-cases $propagate_{NOT}E[elim]$: $propagate_{NOT}\ S\ T$

end

locale *decide-ops* =
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
trail :: $'st \Rightarrow ('v,\ unit)\ ann\text{-}lits$ **and**
$clauses_{NOT}$ :: $'st \Rightarrow 'v\ clauses$ **and**
prepend-trail :: $('v,\ unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$
begin
inductive $decide_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$decide_{NOT}[intro]$: $undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
$\Longrightarrow T \sim prepend\text{-}trail\ (Decided\ L)\ S$
$\Longrightarrow decide_{NOT}\ S\ T$

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

locale *backjumping-ops* =
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
trail :: $'st \Rightarrow ('v,\ unit)\ ann\text{-}lits$ **and**
$clauses_{NOT}$ :: $'st \Rightarrow 'v\ clauses$ **and**
prepend-trail :: $('v,\ unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ +
fixes
backjump-conds :: $'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
begin

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

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

end

5.2.3 DPLL with backjumping

locale *dpll-with-backjumping-ops* =
  propagate-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *propagate-conds* +
  decide-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
  backjumping-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *backjump-conds*
  for
    $trail :: 'st \Rightarrow ('v, unit)\ ann\text{-}lits$ **and**
    $clauses_{NOT} :: 'st \Rightarrow 'v\ clauses$ **and**
    $prepend\text{-}trail :: ('v, unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
    $tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
    $add\text{-}cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
    $remove\text{-}cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
    $inv :: 'st \Rightarrow bool$ **and**
    $backjump\text{-}conds :: 'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **and**
    $propagate\text{-}conds :: ('v, unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow bool$ +
  assumes
      bj-can-jump:
      $\bigwedge S\ C\ F'\ K\ F\ L.$
        $inv\ S \Longrightarrow$
        $no\text{-}dup\ (trail\ S) \Longrightarrow$
        $trail\ S = F'\ @\ Decided\ K\ \#\ F \Longrightarrow$
        $C \in\#\ clauses_{NOT}\ S \Longrightarrow$
        $trail\ S \models as\ CNot\ C \Longrightarrow$
        $undefined\text{-}lit\ F\ L \Longrightarrow$
        $atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F)) \Longrightarrow$
        $clauses_{NOT}\ S \models pm\ C' + \{\#L\#\} \Longrightarrow$
        $F \models as\ CNot\ C' \Longrightarrow$
        $\neg no\text{-}step\ backjump\ S$

begin

We cannot add a like condition $atms\text{-}of\ C' \subseteq atms\text{-}of\text{-}ms\ N$ to ensure that we can backjump even if the last decision variable has disappeared from the set of clauses.

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

Definition

We define dpll with backjumping:

inductive $dpll\text{-}bj :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
$bj\text{-}decide_{NOT}$: $decide_{NOT}\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$ |
$bj\text{-}propagate_{NOT}$: $propagate_{NOT}\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$ |
bj-backjump: $backjump\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$

lemmas *dpll-bj-induct* = *dpll-bj.induct*[*split-format*(*complete*)]
thm *dpll-bj-induct*[*OF dpll-with-backjumping-ops-axioms*]
lemma *dpll-bj-all-induct*[*consumes 2*, *case-names* $decide_{NOT}$ $propagate_{NOT}$ *backjump*]:
  fixes $S\ T :: 'st$
  assumes
    dpll-bj S T **and**
    inv S

$\bigwedge L\ T.\ undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
$\Longrightarrow T \sim prepend\text{-}trail\ (Decided\ L)\ S$
$\Longrightarrow P\ S\ T$ **and**
$\bigwedge C\ L\ T.\ C + \{\#L\#\} \in\#\ clauses_{NOT}\ S \Longrightarrow trail\ S \models as\ CNot\ C \Longrightarrow undefined\text{-}lit\ (trail\ S)\ L$
$\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ S$
$\Longrightarrow P\ S\ T$ **and**
$\bigwedge C\ F'\ K\ F\ L\ C'\ T.\ C \in\#\ clauses_{NOT}\ S \Longrightarrow F'\ @\ Decided\ K\ \#\ F \models as\ CNot\ C$
$\Longrightarrow trail\ S = F'\ @\ Decided\ K\ \#\ F$
$\Longrightarrow undefined\text{-}lit\ F\ L$
$\Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F))$
$\Longrightarrow clauses_{NOT}\ S \models pm\ C' + \{\#L\#\}$
$\Longrightarrow F \models as\ CNot\ C'$
$\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ S)$
$\Longrightarrow P\ S\ T$
shows $P\ S\ T$
⟨*proof*⟩

Basic properties

First, some better suited induction principle **lemma** *dpll-bj-clauses*:
assumes *dpll-bj S T* **and** *inv S*
shows $clauses_{NOT}\ S = clauses_{NOT}\ T$
⟨*proof*⟩

No duplicates in the trail **lemma** *dpll-bj-no-dup*:
assumes *dpll-bj S T* **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
⟨*proof*⟩

Valuations **lemma** *dpll-bj-sat-iff*:
assumes *dpll-bj S T* **and** *inv S*
shows $I \models sm\ clauses_{NOT}\ S \longleftrightarrow I \models sm\ clauses_{NOT}\ T$
⟨*proof*⟩

Clauses **lemma** *dpll-bj-atms-of-ms-clauses-inv*:
assumes
dpll-bj S T **and**
inv S
shows $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) = atms\text{-}of\text{-}mm\ (clauses_{NOT}\ T)$
⟨*proof*⟩

lemma *dpll-bj-atms-in-trail*:
assumes
dpll-bj S T **and**
inv S **and**
$atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
shows $atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ T)) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
⟨*proof*⟩

lemma *dpll-bj-atms-in-trail-in-set*:
assumes *dpll-bj S T***and**
inv S **and**
$atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$ **and**
$atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$

shows *atm-of* ‘ (*lits-of-l* (*trail* T)) ⊆ A
⟨*proof*⟩

lemma *dpll-bj-all-decomposition-implies-inv*:
assumes
dpll-bj S T **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ T) (*get-all-ann-decomposition* (*trail* T))
⟨*proof*⟩

Termination

Using a proper measure **lemma** *length-get-all-ann-decomposition-append-Decided*:
length (*get-all-ann-decomposition* (F' @ *Decided* K # F)) =
length (*get-all-ann-decomposition* F')
+ *length* (*get-all-ann-decomposition* (*Decided* K # F))
− *1*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *length-get-all-ann-decomposition-length*:
length (*get-all-ann-decomposition* M) ≤ *1* + *length* M
⟨*proof*⟩

lemma *length-in-get-all-ann-decomposition-bounded*:
assumes *i*:*i* ∈ *set* (*trail-weight* S)
shows *i* ≤ *Suc* (*length* (*trail* S))
⟨*proof*⟩

Well-foundedness The bounds are the following:

- *1* + *card* (*atms-of-ms* A): *card* (*atms-of-ms* A) is an upper bound on the length of the list. As *get-all-ann-decomposition* appends an possibly empty couple at the end, adding one is needed.
- *2* + *card* (*atms-of-ms* A): *card* (*atms-of-ms* A) is an upper bound on the number of elements, where adding one is necessary for the same reason as for the bound on the list, and one is needed to have a strict bound.

abbreviation *unassigned-lit* :: *'b literal multiset set* ⇒ *'a list* ⇒ *nat* **where**
unassigned-lit N M ≡ *card* (*atms-of-ms* N) − *length* M
lemma *dpll-bj-trail-mes-increasing-prop*:
fixes M :: (*'v*, *unit*) *ann-lits* **and** N :: *'v clauses*
assumes
dpll-bj S T **and**
inv S **and**
NA: *atms-of-mm* (*clauses*$_{NOT}$ S) ⊆ *atms-of-ms* A **and**

MA: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight T*)
> μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight S*)
⟨*proof*⟩

lemma *dpll-bj-trail-mes-decreasing-prop*:
assumes *dpll*: *dpll-bj S T* **and** *inv*: *inv S* **and**
N-A: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
M-A: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
nd: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows (*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*))
− μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight T*)
< (*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*))
− μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight S*)
⟨*proof*⟩

lemma *wf-dpll-bj*:
assumes *fin*: *finite A*
shows *wf* {(*T*, *S*). *dpll-bj S T*
∧ *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*) ∧ *inv S*}
(**is** *wf ?A*)
⟨*proof*⟩

Normal Forms

We prove that given a normal form of DPLL, with some structural invariants, then either N is satisfiable and the built valuation M is a model; or N is unsatisfiable.

Idea of the proof: We have to prove tat *satisfiable* N, ¬ M ⊨*as* N and there is no remaining step is incompatible.

1. The *decide* rule tells us that every variable in N has a value.
2. The assumption ¬ M ⊨*as* N implies that there is conflict.
3. There is at least one decision in the trail (otherwise, M would be a model of the set of clauses N).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

   The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj S*

theorem *dpll-backjump-final-state*:
fixes *A* :: *'v clause set* **and** *S T* :: *'st*
assumes
atms-of-mm ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
no-dup (*trail S*) **and**
finite A **and**
inv: *inv S* **and**

n-s: *no-step dpll-bj S* **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
∨ (*trail S* ⊨*asm clauses*$_{NOT}$ *S* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))
⟨*proof*⟩

end — End of *dpll-with-backjumping-ops*

locale *dpll-with-backjumping* =
dpll-with-backjumping-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *inv*
backjump-conds propagate-conds
for
trail :: '*st* ⇒ ('*v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: '*st* ⇒ '*v clauses* **and**
prepend-trail :: ('*v*, *unit*) *ann-lit* ⇒ '*st* ⇒ '*st* **and**
tl-trail :: '*st* ⇒'*st* **and**
add-cls$_{NOT}$ :: '*v clause* ⇒ '*st* ⇒ '*st* **and**
remove-cls$_{NOT}$ :: '*v clause* ⇒ '*st* ⇒ '*st* **and**
inv :: '*st* ⇒ *bool* **and**
backjump-conds :: '*v clause* ⇒ '*v clause* ⇒ '*v literal* ⇒ '*st* ⇒ '*st* ⇒ *bool* **and**
propagate-conds :: ('*v*, *unit*) *ann-lit* ⇒ '*st* ⇒ *bool*
\+
assumes *dpll-bj-inv*: ⋀*S T*. *dpll-bj S T* ⟹ *inv S* ⟹ *inv T*
begin

lemma *rtranclp-dpll-bj-inv*:
assumes *dpll-bj*** *S T* **and** *inv S*
shows *inv T*
⟨*proof*⟩

lemma *rtranclp-dpll-bj-no-dup*:
assumes *dpll-bj*** *S T* **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
⟨*proof*⟩

lemma *rtranclp-dpll-bj-atms-of-ms-clauses-inv*:
assumes
dpll-bj** *S T* **and** *inv S*
shows *atms-of-mm* (*clauses*$_{NOT}$ *S*) = *atms-of-mm* (*clauses*$_{NOT}$ *T*)
⟨*proof*⟩

lemma *rtranclp-dpll-bj-atms-in-trail*:
assumes
dpll-bj** *S T* **and**
inv S **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *T*)
⟨*proof*⟩

lemma *rtranclp-dpll-bj-sat-iff*:
assumes *dpll-bj*** *S T* **and** *inv S*
shows *I* ⊨*sm clauses*$_{NOT}$ *S* ⟷ *I* ⊨*sm clauses*$_{NOT}$ *T*
⟨*proof*⟩

lemma *rtranclp-dpll-bj-atms-in-trail-in-set*:

assumes
$dpll\text{-}bj^{**}$ S T **and**
inv S
$atms\text{-}of\text{-}mm$ $(clauses_{NOT}\ S) \subseteq A$ **and**
$atm\text{-}of$ ` $(lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$
shows $atm\text{-}of$ ` $(lits\text{-}of\text{-}l\ (trail\ T)) \subseteq A$
⟨*proof*⟩

lemma *rtranclp-dpll-bj-all-decomposition-implies-inv*:
assumes
$dpll\text{-}bj^{**}$ S T **and**
inv S
$all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ S)\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S))$
shows $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ T)\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ T))$
⟨*proof*⟩

lemma *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*:
$\{(T, S).\ dpll\text{-}bj^{++}\ S\ T$
$\wedge\ atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of$ ` $lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
$\wedge\ no\text{-}dup\ (trail\ S) \wedge inv\ S\}$
$\subseteq \{(T, S).\ dpll\text{-}bj\ S\ T \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$
$\wedge\ atm\text{-}of$ ` $lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge no\text{-}dup\ (trail\ S) \wedge inv\ S\}^{+}$
(**is** $?A \subseteq ?B^{+}$)
⟨*proof*⟩

lemma *wf-tranclp-dpll-bj*:
assumes *fin*: *finite* A
shows $wf\ \{(T, S).\ dpll\text{-}bj^{++}\ S\ T$
$\wedge\ atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of$ ` $lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
$\wedge\ no\text{-}dup\ (trail\ S) \wedge inv\ S\}$
⟨*proof*⟩

lemma *dpll-bj-sat-ext-iff*:
$dpll\text{-}bj\ S\ T \Longrightarrow inv\ S \Longrightarrow I{\models}sextm\ clauses_{NOT}\ S \longleftrightarrow I{\models}sextm\ clauses_{NOT}\ T$
⟨*proof*⟩

lemma *rtranclp-dpll-bj-sat-ext-iff*:
$dpll\text{-}bj^{**}\ S\ T \Longrightarrow inv\ S \Longrightarrow I{\models}sextm\ clauses_{NOT}\ S \longleftrightarrow I{\models}sextm\ clauses_{NOT}\ T$
⟨*proof*⟩

theorem *full-dpll-backjump-final-state*:
fixes $A :: 'v\ clause\ set$ **and** $S\ T :: 'st$
assumes
full: *full dpll-bj* S T **and**
atms-S: $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
atms-trail: $atm\text{-}of$ ` $lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
n-d: $no\text{-}dup\ (trail\ S)$ **and**
finite A **and**
inv: $inv\ S$ **and**
decomp: $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ S)\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S))$
shows $unsatisfiable\ (set\text{-}mset\ (clauses_{NOT}\ S))$
$\vee\ (trail\ T \models asm\ clauses_{NOT}\ S \wedge satisfiable\ (set\text{-}mset\ (clauses_{NOT}\ S)))$
⟨*proof*⟩

corollary *full-dpll-backjump-final-state-from-init-state*:
fixes $A :: 'v\ clause\ set$ **and** $S\ T :: 'st$

assumes
full: *full dpll-bj S T* **and**
trail S = [] **and**
$clauses_{NOT}$ *S* = *N* **and**
inv S
shows *unsatisfiable* (*set-mset N*) ∨ (*trail T* ⊨*asm N* ∧ *satisfiable* (*set-mset N*))
⟨*proof*⟩

lemma *tranclp-dpll-bj-trail-mes-decreasing-prop*:
assumes *dpll*: $dpll\text{-}bj^{++}$ *S T* **and** *inv*: *inv S* **and**
N-A: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
M-A: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*)
< (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)
⟨*proof*⟩

end — End of *dpll-with-backjumping*

5.2.4 CDCL

In this section we will now define the conflict driven clause learning above DPLL: we first introduce the rules learn and forget, and the add these rules to the DPLL calculus.

Learn and Forget

Learning adds a new clause where all the literals are already included in the clauses.

locale *learn-ops* =
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* +
fixes
learn-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin
inductive *learn* :: *'st* ⇒ *'st* ⇒ *bool* **where**
$learn_{NOT}$*-rule*: $clauses_{NOT}$ *S* ⊨*pm C* ⟹
atms-of C ⊆ *atms-of-mm* ($clauses_{NOT}$ *S*) ∪ *atm-of* ' (*lits-of-l* (*trail S*)) ⟹
learn-cond C S ⟹
T ∼ $add\text{-}cls_{NOT}$ *C S* ⟹
learn S T
inductive-cases $learn_{NOT}E$: *learn S T*

lemma *learn*-μ_C*-stable*:
assumes *learn S T* **and** *no-dup* (*trail S*)
shows μ_C *A B* (*trail-weight S*) = μ_C *A B* (*trail-weight T*)
⟨*proof*⟩

end

Forget removes an information that can be deduced from the context (e.g. redundant clauses, tautologies)

locale *forget-ops* =
  dpll-state trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
  for
    trail :: $'st \Rightarrow ('v,\ unit)$ *ann-lits* **and**
    clauses$_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
    prepend-trail :: $('v,\ unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
    tl-trail :: $'st \Rightarrow 'st$ **and**
    add-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
    remove-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ +
  fixes
    forget-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow$ *bool*
begin
inductive *forget*$_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow$ *bool* **where**
forget$_{NOT}$:
  removeAll-mset C(*clauses*$_{NOT}$ S) $\models pm$ $C \Longrightarrow$
  forget-cond C $S \Longrightarrow$
  $C \in\#$ *clauses*$_{NOT}$ $S \Longrightarrow$
  $T \sim$ *remove-cls*$_{NOT}$ C $S \Longrightarrow$
  forget$_{NOT}$ S T
inductive-cases *forget*$_{NOT}E$: *forget*$_{NOT}$ S T

lemma *forget-*μ_C*-stable*:
  assumes *forget*$_{NOT}$ S T
  shows μ_C A B (*trail-weight* S) = μ_C A B (*trail-weight* T)
  ⟨*proof*⟩
end

locale *learn-and-forget*$_{NOT}$ =
  learn-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *learn-cond* +
  forget-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *forget-cond*
  for
    trail :: $'st \Rightarrow ('v,\ unit)$ *ann-lits* **and**
    clauses$_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
    prepend-trail :: $('v,\ unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
    tl-trail :: $'st \Rightarrow 'st$ **and**
    add-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
    remove-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
    learn-cond forget-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow$ *bool*
begin
inductive *learn-and-forget*$_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow$ *bool*
where
lf-learn: *learn* S $T \Longrightarrow$ *learn-and-forget*$_{NOT}$ S T |
lf-forget: *forget*$_{NOT}$ S $T \Longrightarrow$ *learn-and-forget*$_{NOT}$ S T
end

Definition of CDCL

locale *conflict-driven-clause-learning-ops* =
  dpll-with-backjumping-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
    inv backjump-conds propagate-conds +
  learn-and-forget$_{NOT}$ *trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *learn-cond*
    forget-cond

for
  trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
  $clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
  prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
  tl-trail :: $'st \Rightarrow 'st$ **and**
  $add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  $remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  inv :: $'st \Rightarrow$ *bool* **and**
  backjump-conds :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow$ *bool* **and**
  propagate-conds :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow$ *bool* **and**
  learn-cond forget-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow$ *bool*
begin

inductive $cdcl_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow$ *bool* **for** S :: $'st$ **where**
c-dpll-bj: *dpll-bj* $S\ S' \Longrightarrow cdcl_{NOT}\ S\ S'$ |
c-learn: *learn* $S\ S' \Longrightarrow cdcl_{NOT}\ S\ S'$ |
$c\text{-}forget_{NOT}$: $forget_{NOT}\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S'$

lemma $cdcl_{NOT}$*-all-induct*[*consumes 1*, *case-names dpll-bj learn* $forget_{NOT}$]:
  fixes $S\ T$ :: $'st$
  assumes $cdcl_{NOT}\ S\ T$ **and**
    dpll: $\bigwedge T.$ *dpll-bj* $S\ T \Longrightarrow P\ S\ T$ **and**
    learning:
      $\bigwedge C\ T.\ clauses_{NOT}\ S \models pm\ C \Longrightarrow$
      atms-of $C \subseteq$ *atms-of-mm* $(clauses_{NOT}\ S) \cup$ *atm-of* ` (*lits-of-l* (*trail* S)) $\Longrightarrow$
      $T \sim add\text{-}cls_{NOT}\ C\ S \Longrightarrow$
      $P\ S\ T$ **and**
    forgetting: $\bigwedge C\ T.$ *removeAll-mset* $C\ (clauses_{NOT}\ S) \models pm\ C \Longrightarrow$
      $C \in\#\ clauses_{NOT}\ S \Longrightarrow$
      $T \sim remove\text{-}cls_{NOT}\ C\ S \Longrightarrow$
      $P\ S\ T$
  shows $P\ S\ T$
  ⟨*proof*⟩

lemma $cdcl_{NOT}$*-no-dup*:
  assumes
    $cdcl_{NOT}\ S\ T$ **and**
    inv S **and**
    no-dup (*trail* S)
  shows *no-dup* (*trail* T)
  ⟨*proof*⟩

Consistency of the trail **lemma** $cdcl_{NOT}$*-consistent*:
  assumes
    $cdcl_{NOT}\ S\ T$ **and**
    inv S **and**
    no-dup (*trail* S)
  shows *consistent-interp* (*lits-of-l* (*trail* T))
  ⟨*proof*⟩

The subtle problem here is that tautologies can be removed, meaning that some variable can disappear of the problem. It is also means that some variable of the trail might not be present in the clauses anymore.

lemma $cdcl_{NOT}$*-atms-of-ms-clauses-decreasing*:
  assumes $cdcl_{NOT}\ S\ T$**and** *inv* S **and** *no-dup* (*trail* S)

shows *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-atms-in-trail*:
assumes *cdcl*$_{NOT}$ *S T***and** *inv S* **and** *no-dup* (*trail S*)
and *atm-of* ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-atms-in-trail-in-set*:
assumes
cdcl$_{NOT}$ *S T* **and** *inv S* **and** *no-dup* (*trail S*) **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *A*
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-all-decomposition-implies*:
assumes *cdcl*$_{NOT}$ *S T* **and** *inv S* **and** *n-d*[*simp*]: *no-dup* (*trail S*) **and**
all-decomposition-implies-m (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
⟨*proof*⟩

Extension of models

lemma *cdcl*$_{NOT}$*-bj-sat-ext-iff*:
assumes *cdcl*$_{NOT}$ *S T***and** *inv S* **and** *n-d*: *no-dup* (*trail S*)
shows *I*⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I*⊨*sextm clauses*$_{NOT}$ *T*
⟨*proof*⟩

end — end of *conflict-driven-clause-learning-ops*

CDCL with invariant

locale *conflict-driven-clause-learning* =
conflict-driven-clause-learning-ops +
assumes *cdcl*$_{NOT}$*-inv*: ⋀*S T*. *cdcl*$_{NOT}$ *S T* ⟹ *inv S* ⟹ *inv T*
begin
sublocale *dpll-with-backjumping*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-inv*:
cdcl$_{NOT}$** *S T* ⟹ *inv S* ⟹ *inv T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-no-dup*:
assumes *cdcl*$_{NOT}$** *S T* **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-trail-clauses-bound*:
assumes
cdcl: *cdcl*$_{NOT}$** *S T* **and**
inv: *inv S* **and**
n-d: *no-dup* (*trail S*) **and**
atms-clauses-S: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**

atms-trail-S: $atm\text{-}of$ ‘($lits\text{-}of\text{-}l$ ($trail$ S)) $\subseteq A$
shows $atm\text{-}of$ ‘ ($lits\text{-}of\text{-}l$ ($trail$ T)) $\subseteq A \wedge atms\text{-}of\text{-}mm$ ($clauses_{NOT}$ T) $\subseteq A$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-all-decomposition-implies*:
assumes $cdcl_{NOT}{}^{**}$ S T **and** *inv* S **and** *no-dup* (*trail* S) **and**
all-decomposition-implies-m ($clauses_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows
all-decomposition-implies-m ($clauses_{NOT}$ T) (*get-all-ann-decomposition* (*trail* T))
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-bj-sat-ext-iff*:
assumes $cdcl_{NOT}{}^{**}$ S T**and** *inv* S **and** *no-dup* (*trail* S)
shows $I \models sextm\ clauses_{NOT}\ S \longleftrightarrow I \models sextm\ clauses_{NOT}\ T$
⟨*proof*⟩

definition *cdcl*$_{NOT}$*-NOT-all-inv* **where**
$cdcl_{NOT}$*-NOT-all-inv* A $S \longleftrightarrow$ (*finite* $A \wedge$ *inv* $S \wedge$ *atms-of-mm* ($clauses_{NOT}$ S) $\subseteq$ *atms-of-ms* A
$\wedge$ *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* $A \wedge$ *no-dup* (*trail* S))

lemma *cdcl*$_{NOT}$*-NOT-all-inv*:
assumes $cdcl_{NOT}{}^{**}$ S T **and** $cdcl_{NOT}$*-NOT-all-inv* A S
shows $cdcl_{NOT}$*-NOT-all-inv* A T
⟨*proof*⟩

abbreviation *learn-or-forget* **where**
learn-or-forget S $T \equiv$ *learn* S $T \vee forget_{NOT}$ S T

lemma *rtranclp-learn-or-forget-cdcl*$_{NOT}$:
learn-or-forget** S $T \Longrightarrow cdcl_{NOT}{}^{**}$ S T
⟨*proof*⟩

lemma *learn-or-forget-dpll-*μ_C:
assumes
l-f: *learn-or-forget*** S T **and**
dpll: *dpll-bj* T U **and**
inv: $cdcl_{NOT}$*-NOT-all-inv* A S
shows (*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))
$-$ μ_C (*1*+*card* (*atms-of-ms* A)) (*2*+*card* (*atms-of-ms* A)) (*trail-weight* U)
$<$ (*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))
$-$ μ_C (*1*+*card* (*atms-of-ms* A)) (*2*+*card* (*atms-of-ms* A)) (*trail-weight* S)
(**is** $?\mu$ U $<$ $?\mu$ S)
⟨*proof*⟩

lemma *infinite-cdcl*$_{NOT}$*-exists-learn-and-forget-infinite-chain*:
assumes
$\bigwedge i.\ cdcl_{NOT}\ (f\ i)\ (f(Suc\ i))$ **and**
inv: $cdcl_{NOT}$*-NOT-all-inv* A (f *0*)
shows $\exists j.\ \forall i \geq j.$ *learn-or-forget* (f i) (f (*Suc* i))
⟨*proof*⟩

lemma *wf-cdcl*$_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge f\ j.\ \neg\ (\forall i \geq j.$ *learn-or-forget* (f i) (f (*Suc* i)))
shows *wf* $\{(T, S).\ cdcl_{NOT}\ S\ T \wedge cdcl_{NOT}$*-NOT-all-inv* A $S\}$

(**is** *wf* $\{(T, S).\ cdcl_{NOT}\ S\ T \wedge ?inv\ S\}$)
⟨*proof*⟩

lemma *inv-and-tranclp-cdcl-$_{NOT}$-tranclp-cdcl$_{NOT}$-and-inv*:
$cdcl_{NOT}{}^{++}\ S\ T \wedge cdcl_{NOT}\text{-}NOT\text{-}all\text{-}inv\ A\ S \longleftrightarrow (\lambda S\ T.\ cdcl_{NOT}\ S\ T \wedge cdcl_{NOT}\text{-}NOT\text{-}all\text{-}inv\ A\ S)^{++}\ S\ T$
(**is** $?A \wedge ?I \longleftrightarrow ?B$)
⟨*proof*⟩

lemma *wf-tranclp-cdcl$_{NOT}$-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge f\ j.\ \neg\ (\forall\ i{\geq}j.\ learn\text{-}or\text{-}forget\ (f\ i)\ (f\ (Suc\ i)))$
shows *wf* $\{(T, S).\ cdcl_{NOT}{}^{++}\ S\ T \wedge cdcl_{NOT}\text{-}NOT\text{-}all\text{-}inv\ A\ S\}$
⟨*proof*⟩

lemma *cdcl$_{NOT}$-final-state*:
assumes
n-s: *no-step* $cdcl_{NOT}\ S$ **and**
inv: $cdcl_{NOT}\text{-}NOT\text{-}all\text{-}inv\ A\ S$ **and**
decomp: *all-decomposition-implies-m* $(clauses_{NOT}\ S)$ (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* $(clauses_{NOT}\ S)$)
$\vee$ (*trail* $S \models asm\ clauses_{NOT}\ S \wedge$ *satisfiable* (*set-mset* $(clauses_{NOT}\ S)$))
⟨*proof*⟩

lemma *full-cdcl$_{NOT}$-final-state*:
assumes
full: *full* $cdcl_{NOT}\ S\ T$ **and**
inv: $cdcl_{NOT}\text{-}NOT\text{-}all\text{-}inv\ A\ S$ **and**
n-d: *no-dup* (*trail* S) **and**
decomp: *all-decomposition-implies-m* $(clauses_{NOT}\ S)$ (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* $(clauses_{NOT}\ T)$)
$\vee$ (*trail* $T \models asm\ clauses_{NOT}\ T \wedge$ *satisfiable* (*set-mset* $(clauses_{NOT}\ T)$))
⟨*proof*⟩

end — end of *conflict-driven-clause-learning*

Termination

To prove termination we need to restrict learn and forget. Otherwise we could forget and relearn the exact same clause over and over. A first idea is to forbid removing clauses that can be used to backjump. This does not change the rules of the calculus. A second idea is to "merge" backjump and learn: that way, though closer to implementation, needs a change of the rules, since the backjump-rule learns the clause used to backjump.

Restricting learn and forget

locale *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
conflict-driven-clause-learning trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds
$\lambda C\ S.\ distinct\text{-}mset\ C \wedge \neg tautology\ C \wedge learn\text{-}restrictions\ C\ S\ \wedge$
$(\exists F\ K\ d\ F'\ C'\ L.\ trail\ S = F'\ @\ Decided\ K\ \#\ F \wedge C = C' + \{\#L\#\} \wedge F \models as\ CNot\ C'$
$\wedge\ C' + \{\#L\#\} \notin\# clauses_{NOT}\ S)$
$\lambda C\ S.\ \neg(\exists F'\ F\ K\ d\ L.\ trail\ S = F'\ @\ Decided\ K\ \#\ F \wedge F \models as\ CNot\ (remove1\text{-}mset\ L\ C))$
$\wedge\ forget\text{-}restrictions\ C\ S$

for
$trail$ :: $'st \Rightarrow ('v,\ unit)\ ann\text{-}lits$ **and**
$clauses_{NOT}$ :: $'st \Rightarrow\ 'v\ clauses$ **and**
$prepend\text{-}trail$ :: $('v,\ unit)\ ann\text{-}lit \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
$tl\text{-}trail$ :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
inv :: $'st \Rightarrow bool$ **and**
$backjump\text{-}conds$ :: $'v\ clause \Rightarrow\ 'v\ clause \Rightarrow\ 'v\ literal \Rightarrow\ 'st \Rightarrow\ 'st \Rightarrow bool$ **and**
$propagate\text{-}conds$ :: $('v,\ unit)\ ann\text{-}lit \Rightarrow\ 'st \Rightarrow bool$ **and**
$learn\text{-}restrictions\ forget\text{-}restrictions$ :: $'v\ clause \Rightarrow\ 'st \Rightarrow bool$
begin

lemma $cdcl_{NOT}\text{-}learn\text{-}all\text{-}induct[consumes\ 1,\ case\text{-}names\ dpll\text{-}bj\ learn\ forget_{NOT}]$:
fixes $S\ T$ :: $'st$
assumes $cdcl_{NOT}\ S\ T$ **and**
$dpll$: $\bigwedge T.\ dpll\text{-}bj\ S\ T \Longrightarrow P\ S\ T$ **and**
$learning$:
$\bigwedge C\ F\ K\ F'\ C'\ L\ T.\ clauses_{NOT}\ S \models pm\ C \Longrightarrow$
$atms\text{-}of\ C \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S)) \Longrightarrow$
$distinct\text{-}mset\ C \Longrightarrow$
$\neg\ tautology\ C \Longrightarrow$
$learn\text{-}restrictions\ C\ S \Longrightarrow$
$trail\ S = F'\ @\ Decided\ K\ \#\ F \Longrightarrow$
$C = C' + \{\#L\#\} \Longrightarrow$
$F \models as\ CNot\ C' \Longrightarrow$
$C' + \{\#L\#\} \notin\#\ clauses_{NOT}\ S \Longrightarrow$
$T \sim add\text{-}cls_{NOT}\ C\ S \Longrightarrow$
$P\ S\ T$ **and**
$forgetting$: $\bigwedge C\ T.\ removeAll\text{-}mset\ C\ (clauses_{NOT}\ S) \models pm\ C \Longrightarrow$
$C \in\#\ clauses_{NOT}\ S \Longrightarrow$
$\neg(\exists F'\ F\ K\ L.\ trail\ S = F'\ @\ Decided\ K\ \#\ F \land F \models as\ CNot\ (C - \{\#L\#\})) \Longrightarrow$
$T \sim remove\text{-}cls_{NOT}\ C\ S \Longrightarrow$
$forget\text{-}restrictions\ C\ S \Longrightarrow$
$P\ S\ T$
shows $P\ S\ T$
$\langle proof \rangle$

lemma $rtranclp\text{-}cdcl_{NOT}\text{-}inv$:
$cdcl_{NOT}^{**}\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
$\langle proof \rangle$

lemma $learn\text{-}always\text{-}simple\text{-}clauses$:
assumes
$learn$: $learn\ S\ T$ **and**
$n\text{-}d$: $no\text{-}dup\ (trail\ S)$
shows $set\text{-}mset\ (clauses_{NOT}\ T - clauses_{NOT}\ S)$
$\subseteq simple\text{-}clss\ (atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S))$
$\langle proof \rangle$

definition $conflicting\text{-}bj\text{-}clss\ S \equiv$
$\{C+\{\#L\#\}\ |C\ L.\ C+\{\#L\#\} \in\#\ clauses_{NOT}\ S \land distinct\text{-}mset\ (C+\{\#L\#\})$
$\land \neg tautology\ (C+\{\#L\#\})$
$\land (\exists F'\ K\ F.\ trail\ S = F'\ @\ Decided\ K\ \#\ F \land F \models as\ CNot\ C)\}$

lemma $conflicting\text{-}bj\text{-}clss\text{-}remove\text{-}cls_{NOT}[simp]$:

conflicting-bj-clss (*remove-cls$_{NOT}$* *C S*) = *conflicting-bj-clss* *S* − {*C*}
⟨*proof*⟩

lemma *conflicting-bj-clss-remove-cls$_{NOT}$′*[*simp*]:
T ∼ *remove-cls$_{NOT}$* *C S* ⟹ *conflicting-bj-clss* *T* = *conflicting-bj-clss* *S* − {*C*}
⟨*proof*⟩

lemma *conflicting-bj-clss-add-cls$_{NOT}$-state-eq*:
assumes
T: *T* ∼ *add-cls$_{NOT}$* *C′ S* **and**
n-d: *no-dup* (*trail S*)
shows *conflicting-bj-clss T*
= *conflicting-bj-clss S*
∪ (*if* ∃ *C L*. *C′* = *C* +{#*L*#} ∧ *distinct-mset* (*C*+{#*L*#}) ∧ ¬*tautology* (*C*+{#*L*#})
∧ (∃ *F′ K d F*. *trail S* = *F′* @ *Decided K* # *F* ∧ *F* ⊨*as CNot C*)
then {*C′*} *else* {})
⟨*proof*⟩

lemma *conflicting-bj-clss-add-cls$_{NOT}$*:
no-dup (*trail S*) ⟹
conflicting-bj-clss (*add-cls$_{NOT}$* *C′ S*)
= *conflicting-bj-clss S*
∪ (*if* ∃ *C L*. *C′* = *C* +{#*L*#}∧ *distinct-mset* (*C*+{#*L*#}) ∧ ¬*tautology* (*C*+{#*L*#})
∧ (∃ *F′ K d F*. *trail S* = *F′* @ *Decided K* # *F* ∧ *F* ⊨*as CNot C*)
then {*C′*} *else* {})
⟨*proof*⟩

lemma *conflicting-bj-clss-incl-clauses*:
conflicting-bj-clss S ⊆ *set-mset* (*clauses$_{NOT}$* *S*)
⟨*proof*⟩

lemma *finite-conflicting-bj-clss*[*simp*]:
finite (*conflicting-bj-clss S*)
⟨*proof*⟩

lemma *learn-conflicting-increasing*:
no-dup (*trail S*) ⟹ *learn S T* ⟹ *conflicting-bj-clss S* ⊆ *conflicting-bj-clss T*
⟨*proof*⟩

abbreviation *conflicting-bj-clss-yet b S* ≡
3 ^ *b* − *card* (*conflicting-bj-clss S*)

abbreviation μ_L :: *nat* ⇒ *′st* ⇒ *nat* × *nat* **where**
μ_L *b S* ≡ (*conflicting-bj-clss-yet b S*, *card* (*set-mset* (*clauses$_{NOT}$* *S*)))

lemma *remove1-mset-single-add-if*:
remove1-mset L (*C* + {#*L′*#}) = (*if L* = *L′ then C else remove1-mset L C* + {#*L′*#})
⟨*proof*⟩

lemma *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*:
assumes *forget$_{NOT}$* *S T*
shows *conflicting-bj-clss S* = *conflicting-bj-clss T*
⟨*proof*⟩

lemma *forget-μ_L-decrease*:
assumes *forget$_{NOT}$*: *forget$_{NOT}$* *S T*

shows $(\mu_L\ b\ T,\ \mu_L\ b\ S) \in$ *less-than* <*lex*> *less-than*
⟨*proof*⟩

lemma *set-condition-or-split*:
$\{a.\ (a = b \vee Q\ a) \wedge S\ a\} = (\mathit{if}\ S\ b\ \mathit{then}\ \{b\}\ \mathit{else}\ \{\}) \cup \{a.\ Q\ a \wedge S\ a\}$
⟨*proof*⟩

lemma *set-insert-neq*:
$A \neq \mathit{insert}\ a\ A \longleftrightarrow a \notin A$
⟨*proof*⟩

lemma *learn-μ_L-decrease*:
assumes *learnST*: *learn S T* **and** *n-d*: *no-dup* (*trail S*) **and**
A: *atms-of-mm* ($\mathit{clauses}_{NOT}$ *S*) $\cup$ *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *A* **and**
fin-A: *finite A*
shows $(\mu_L\ (\mathit{card}\ A)\ T,\ \mu_L\ (\mathit{card}\ A)\ S) \in$ *less-than* <*lex*> *less-than*
⟨*proof*⟩

We have to assume the following:

- *inv S*: the invariant holds in the inital state.
- *A* is a (finite *finite A*) superset of the literals in the trail *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* and in the clauses *atms-of-mm* ($\mathit{clauses}_{NOT}$ *S*) $\subseteq$ *atms-of-ms A*. This can the the set of all the literals in the starting set of clauses.
- *no-dup* (*trail S*): no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} **where**
μ_{CDCL} *A T* $\equiv$ ((*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*),
conflicting-bj-clss-yet (*card* (*atms-of-ms A*)) *T*, *card* (*set-mset* ($\mathit{clauses}_{NOT}$ *T*)))
lemma cdcl_{NOT}*-decreasing-measure*:
assumes
cdcl_{NOT} *S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* ($\mathit{clauses}_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
atm-lits: *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows (μ_{CDCL} *A T*, μ_{CDCL} *A S*)
$\in$ *less-than* <*lex*> (*less-than* <*lex*> *less-than*)
⟨*proof*⟩

lemma *wf-*cdcl_{NOT}*-restricted-learning*:
assumes *finite A*
shows *wf* {(*T*, *S*).
(*atms-of-mm* ($\mathit{clauses}_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* $\wedge$ *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *no-dup* (*trail S*)
$\wedge$ *inv S*)
$\wedge$ cdcl_{NOT} *S T* }
⟨*proof*⟩

definition $\mu_C{}'$ :: *'v clause set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_C{}'$ *A T* $\equiv$ μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*)

definition $\mu_{CDCL}{}'$ :: *'v clause set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_{CDCL}{}'$ *A T* $\equiv$
((*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*)) − $\mu_C{}'$ *A T*) ∗ (*1*+ *3*^*card* (*atms-of-ms A*)) ∗
2
+ *conflicting-bj-clss-yet* (*card* (*atms-of-ms A*)) *T* ∗ *2*
+ *card* (*set-mset* (*clauses*$_{NOT}$ *T*))

lemma *cdcl*$_{NOT}$*-decreasing-measure'*:
assumes
cdcl$_{NOT}$ *S T* **and**
inv: *inv S* **and**
atms-clss: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows $\mu_{CDCL}{}'$ *A T* < $\mu_{CDCL}{}'$ *A S*
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-clauses-bound*:
assumes
cdcl$_{NOT}$ *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) ⊆ *A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A[*simp*]: *finite A*
shows *set-mset* (*clauses*$_{NOT}$ *T*) ⊆ *set-mset* (*clauses*$_{NOT}$ *S*) ∪ *simple-clss A*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-clauses-bound*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) ⊆ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *set-mset* (*clauses*$_{NOT}$ *T*) ⊆ *set-mset* (*clauses*$_{NOT}$ *S*) ∪ *simple-clss A*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-card-clauses-bound*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) ⊆ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* (*set-mset* (*clauses*$_{NOT}$ *T*)) ≤ *card* (*set-mset* (*clauses*$_{NOT}$ *S*)) + *3* ^ (*card A*)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-card-clauses-bound'*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**

$atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$ **and**
$atm\text{-}of\ `(lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$ **and**
$n\text{-}d$: $no\text{-}dup\ (trail\ S)$ **and**
$finite$: $finite\ A$
shows $card\ \{C|C.\ C \in\#\ clauses_{NOT}\ T \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\}$
$\leq card\ \{C|C.\ C{\in}\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ (card\ A)$
(**is** $card\ ?T \leq card\ ?S + \text{-}$)
$\langle proof \rangle$

lemma $rtranclp\text{-}cdcl_{NOT}\text{-}card\text{-}simple\text{-}clauses\text{-}bound$:
assumes
$cdcl_{NOT}^{**}\ S\ T$ **and**
$inv\ S$ **and**
NA: $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$ **and**
MA: $atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$ **and**
$n\text{-}d$: $no\text{-}dup\ (trail\ S)$ **and**
$finite$: $finite\ A$
shows $card\ (set\text{-}mset\ (clauses_{NOT}\ T))$
$\leq card\ \{C.\ C \in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ (card\ A)$
(**is** $card\ ?T \leq card\ ?S + \text{-}$)
$\langle proof \rangle$

definition $\mu_{CDCL}'\text{-}bound :: 'v\ clause\ set \Rightarrow 'st \Rightarrow nat$ **where**
$\mu_{CDCL}'\text{-}bound\ A\ S =$
$((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A))) * (1 + 3\ \hat{}\ card\ (atms\text{-}of\text{-}ms\ A)) * 2$
$+ 2{*}3\ \hat{}\ (card\ (atms\text{-}of\text{-}ms\ A))$
$+ card\ \{C.\ C \in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ (card\ (atms\text{-}of\text{-}ms\ A))$

lemma $\mu_{CDCL}'\text{-}bound\text{-}reduce\text{-}trail\text{-}to_{NOT}[simp]$:
$\mu_{CDCL}'\text{-}bound\ A\ (reduce\text{-}trail\text{-}to_{NOT}\ M\ S) = \mu_{CDCL}'\text{-}bound\ A\ S$
$\langle proof \rangle$

lemma $rtranclp\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-}bound\text{-}reduce\text{-}trail\text{-}to_{NOT}$:
assumes
$cdcl_{NOT}^{**}\ S\ T$ **and**
$inv\ S$ **and**
$atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
$atm\text{-}of\ `(lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
$n\text{-}d$: $no\text{-}dup\ (trail\ S)$ **and**
$finite$: $finite\ (atms\text{-}of\text{-}ms\ A)$ **and**
U: $U \sim reduce\text{-}trail\text{-}to_{NOT}\ M\ T$
shows $\mu_{CDCL}'\ A\ U \leq \mu_{CDCL}'\text{-}bound\ A\ S$
$\langle proof \rangle$

lemma $rtranclp\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-}bound$:
assumes
$cdcl_{NOT}^{**}\ S\ T$ **and**
$inv\ S$ **and**
$atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
$atm\text{-}of\ `(lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
$n\text{-}d$: $no\text{-}dup\ (trail\ S)$ **and**
$finite$: $finite\ (atms\text{-}of\text{-}ms\ A)$
shows $\mu_{CDCL}'\ A\ T \leq \mu_{CDCL}'\text{-}bound\ A\ S$
$\langle proof \rangle$

lemma $rtranclp\text{-}\mu_{CDCL}'\text{-}bound\text{-}decreasing$:

assumes
$cdcl_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*$clauses_{NOT}$ S*) $\subseteq$ *atms-of-ms A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite[*simp*]: *finite* (*atms-of-ms A*)
shows μ_{CDCL}′*-bound A T* $\leq$ μ_{CDCL}′*-bound A S*
⟨*proof*⟩

end — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt*

5.2.5 CDCL with restarts

Definition

locale *restart-ops* =
fixes
$cdcl_{NOT}$ :: ′*st* ⇒ ′*st* ⇒ *bool* **and**
restart :: ′*st* ⇒ ′*st* ⇒ *bool*
begin
inductive *$cdcl_{NOT}$-raw-restart* :: ′*st* ⇒ ′*st* ⇒ *bool* **where**
$cdcl_{NOT}$ S T ⟹ *$cdcl_{NOT}$-raw-restart S T* |
restart S T ⟹ *$cdcl_{NOT}$-raw-restart S T*

end

locale *conflict-driven-clause-learning-with-restarts* =
conflict-driven-clause-learning trail $clauses_{NOT}$ prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT}
inv backjump-conds propagate-conds learn-cond forget-cond
for
trail :: ′*st* ⇒ (′*v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: ′*st* ⇒ ′*v clauses* **and**
prepend-trail :: (′*v*, *unit*) *ann-lit* ⇒ ′*st* ⇒ ′*st* **and**
tl-trail :: ′*st* ⇒′*st* **and**
add-cls_{NOT} :: ′*v clause* ⇒ ′*st* ⇒ ′*st* **and**
remove-cls_{NOT} :: ′*v clause* ⇒ ′*st* ⇒ ′*st* **and**
inv :: ′*st* ⇒ *bool* **and**
backjump-conds :: ′*v clause* ⇒ ′*v clause* ⇒ ′*v literal* ⇒ ′*st* ⇒ ′*st* ⇒ *bool* **and**
propagate-conds :: (′*v*, *unit*) *ann-lit* ⇒ ′*st* ⇒ *bool* **and**
learn-cond forget-cond :: ′*v clause* ⇒ ′*st* ⇒ *bool*
begin

lemma *$cdcl_{NOT}$-iff-$cdcl_{NOT}$-raw-restart-no-restarts*:
$cdcl_{NOT}$ S T ⟷ *restart-ops.$cdcl_{NOT}$-raw-restart $cdcl_{NOT}$* (λ- -. *False*) *S T*
(**is** *?C S T* ⟷ *?R S T*)
⟨*proof*⟩

lemma *$cdcl_{NOT}$-$cdcl_{NOT}$-raw-restart*:
$cdcl_{NOT}$ S T ⟹ *restart-ops.$cdcl_{NOT}$-raw-restart $cdcl_{NOT}$ restart S T*
⟨*proof*⟩
end

Increasing restarts

To add restarts we needs some assumptions on the predicate (called *$cdcl_{NOT}$* here):

- a function f that is strictly monotonic. The first step is actually only used as a restart to clean the state (e.g. to ensure that the trail is empty). Then we assume that $(1::'a) \leq f\ n$ for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...

- a measure μ: it should decrease under the assumptions *bound-inv*, whenever a $cdcl_{NOT}$ or a *restart* is done. A parameter is given to μ: for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.

- we also assume that the measure decrease after any $cdcl_{NOT}$ step.

- an invariant on the states $cdcl_{NOT}$*-inv* that also holds after restarts.

- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ*-bound* taking the same parameter as μ and the initial state of the considered $cdcl_{NOT}$ chain.

locale $cdcl_{NOT}$*-increasing-restarts-ops* =
  restart-ops $cdcl_{NOT}$ *restart* **for**
    restart :: *'st* ⇒ *'st* ⇒ *bool* **and**
    $cdcl_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* +
  fixes
    f :: *nat* ⇒ *nat* **and**
    bound-inv :: *'bound* ⇒ *'st* ⇒ *bool* **and**
    μ :: *'bound* ⇒ *'st* ⇒ *nat* **and**
    $cdcl_{NOT}$*-inv* :: *'st* ⇒ *bool* **and**
    μ*-bound* :: *'bound* ⇒ *'st* ⇒ *nat*
  assumes
    f: *unbounded f* **and**
    f-ge-1: $\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \neq 0$ **and**
    bound-inv: $\bigwedge A\ S\ T.\ cdcl_{NOT}\text{-}inv\ S \Longrightarrow bound\text{-}inv\ A\ S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow bound\text{-}inv\ A\ T$ **and**
    $cdcl_{NOT}$*-measure*: $\bigwedge A\ S\ T.\ cdcl_{NOT}\text{-}inv\ S \Longrightarrow bound\text{-}inv\ A\ S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow \mu\ A\ T < \mu\ A\ S$ **and**
    measure-bound2: $\bigwedge A\ T\ U.\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T \Longrightarrow cdcl_{NOT}{}^{**}\ T\ U$
      $\Longrightarrow \mu\ A\ U \leq \mu\text{-}bound\ A\ T$ **and**
    measure-bound4: $\bigwedge A\ T\ U.\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T \Longrightarrow cdcl_{NOT}{}^{**}\ T\ U$
      $\Longrightarrow \mu\text{-}bound\ A\ U \leq \mu\text{-}bound\ A\ T$ **and**
    $cdcl_{NOT}$*-restart-inv*: $\bigwedge A\ U\ V.\ cdcl_{NOT}\text{-}inv\ U \Longrightarrow restart\ U\ V \Longrightarrow bound\text{-}inv\ A\ U \Longrightarrow bound\text{-}inv\ A\ V$
      and
    exists-bound: $\bigwedge R\ S.\ cdcl_{NOT}\text{-}inv\ R \Longrightarrow restart\ R\ S \Longrightarrow \exists\ A.\ bound\text{-}inv\ A\ S$ **and**
    $cdcl_{NOT}$*-inv*: $\bigwedge S\ T.\ cdcl_{NOT}\text{-}inv\ S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow cdcl_{NOT}\text{-}inv\ T$ **and**
    $cdcl_{NOT}$*-inv-restart*: $\bigwedge S\ T.\ cdcl_{NOT}\text{-}inv\ S \Longrightarrow restart\ S\ T \Longrightarrow cdcl_{NOT}\text{-}inv\ T$
begin

lemma $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*:
  assumes
    $(cdcl_{NOT}\widehat{\ }\widehat{\ }n)\ S\ T$ **and**
    $cdcl_{NOT}$*-inv S*
  shows $cdcl_{NOT}$*-inv T*
  ⟨*proof*⟩

lemma $cdcl_{NOT}$*-bound-inv*:
  assumes

$(cdcl_{NOT}\widehat{\ }\widehat{\ }n)$ S T **and**
$cdcl_{NOT}$-*inv* S
bound-inv A S
shows *bound-inv* A T
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-cdcl*$_{NOT}$*-inv*:
assumes
$cdcl_{NOT}{}^{**}$ S T **and**
$cdcl_{NOT}$-*inv* S
shows $cdcl_{NOT}$-*inv* T
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-bound-inv*:
assumes
$cdcl_{NOT}{}^{**}$ S T **and**
bound-inv A S **and**
$cdcl_{NOT}$-*inv* S
shows *bound-inv* A T
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-comp-n-le*:
assumes
$(cdcl_{NOT}\widehat{\ }\widehat{\ }(Suc\ n))$ S T **and**
bound-inv A S
$cdcl_{NOT}$-*inv* S
shows $\mu\ A\ T < \mu\ A\ S - n$
⟨*proof*⟩

lemma *wf-cdcl*$_{NOT}$:
wf $\{(T, S).\ cdcl_{NOT}\ S\ T \wedge cdcl_{NOT}\text{-}inv\ S \wedge bound\text{-}inv\ A\ S\}$ (**is** *wf ?A*)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-measure*:
assumes
$cdcl_{NOT}{}^{**}$ S T **and**
bound-inv A S **and**
$cdcl_{NOT}$-*inv* S
shows $\mu\ A\ T \leq \mu\ A\ S$
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-comp-bounded*:
assumes
bound-inv A S **and** $cdcl_{NOT}$-*inv* S **and** $m \geq 1 + \mu\ A\ S$
shows $\neg(cdcl_{NOT}\ \widehat{\ }\widehat{\ }\ m)$ S T
⟨*proof*⟩

- $f\ n < m$ ensures that at least one step has been done.

inductive *cdcl*$_{NOT}$*-restart* **where**
restart-step: $(cdcl_{NOT}\widehat{\ }\widehat{\ }m)\ S\ T \Longrightarrow m \geq f\ n \Longrightarrow restart\ T\ U$
$\Longrightarrow cdcl_{NOT}\text{-}restart\ (S, n)\ (U, Suc\ n)$ |
restart-full: *full1* $cdcl_{NOT}\ S\ T \Longrightarrow cdcl_{NOT}\text{-}restart\ (S, n)\ (T, Suc\ n)$

lemmas *cdcl*$_{NOT}$*-with-restart-induct* = *cdcl*$_{NOT}$*-restart.induct*[*split-format*(*complete*), *OF* $cdcl_{NOT}$*-increasing-restarts-ops-axioms*]

lemma $cdcl_{NOT}$*-restart-*$cdcl_{NOT}$*-raw-restart*:
$cdcl_{NOT}$*-restart S T* $\Longrightarrow$ $cdcl_{NOT}$*-raw-restart*** (*fst S*) (*fst T*)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-with-restart-bound-inv*:
assumes
$cdcl_{NOT}$*-restart S T* **and**
bound-inv A (*fst S*) **and**
$cdcl_{NOT}$*-inv* (*fst S*)
shows *bound-inv A* (*fst T*)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*:
assumes
$cdcl_{NOT}$*-restart S T* **and**
$cdcl_{NOT}$*-inv* (*fst S*)
shows $cdcl_{NOT}$*-inv* (*fst T*)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*:
assumes
$cdcl_{NOT}$*-restart*** *S T* **and**
$cdcl_{NOT}$*-inv* (*fst S*)
shows $cdcl_{NOT}$*-inv* (*fst T*)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-with-restart-bound-inv*:
assumes
$cdcl_{NOT}$*-restart*** *S T* **and**
$cdcl_{NOT}$*-inv* (*fst S*) **and**
bound-inv A (*fst S*)
shows *bound-inv A* (*fst T*)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-with-restart-increasing-number*:
$cdcl_{NOT}$*-restart S T* $\Longrightarrow$ *snd T* = *1* + *snd S*
⟨*proof*⟩
end

locale $cdcl_{NOT}$*-increasing-restarts* =
$cdcl_{NOT}$*-increasing-restarts-ops restart* $cdcl_{NOT}$ *f bound-inv* μ $cdcl_{NOT}$*-inv* μ*-bound* +
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
f :: *nat* ⇒ *nat* **and**
restart :: *'st* ⇒ *'st* ⇒ *bool* **and**
bound-inv :: *'bound* ⇒ *'st* ⇒ *bool* **and**
μ :: *'bound* ⇒ *'st* ⇒ *nat* **and**
$cdcl_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **and**

$cdcl_{NOT}$*-inv* :: *'st* ⇒ *bool* **and**
μ*-bound* :: *'bound* ⇒ *'st* ⇒ *nat* +
assumes
measure-bound: $\bigwedge A\ T\ V\ n.$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow$ $cdcl_{NOT}$*-restart* $(T,\ n)\ (V,\ Suc\ n) \Longrightarrow \mu\ A\ V \leq \mu$*-bound* $A\ T$ **and**
$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}$*-restart* $(T,\ a)\ (V,\ b) \Longrightarrow$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu$*-bound* $A\ V \leq \mu$*-bound* $A\ T$
begin

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}$*-restart*** $(T,\ a)\ (V,\ b) \Longrightarrow$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu$*-bound* $A\ V \leq \mu$*-bound* $A\ T$
⟨*proof*⟩

lemma $cdcl_{NOT}$*-raw-restart-measure-bound*:
$cdcl_{NOT}$*-restart* $(T,\ a)\ (V,\ b) \Longrightarrow$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu$*-bound* $A\ T$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-measure-bound*:
$cdcl_{NOT}$*-restart*** $(T,\ a)\ (V,\ b) \Longrightarrow$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu$*-bound* $A\ T$
⟨*proof*⟩

lemma *wf-*$cdcl_{NOT}$*-restart*:
wf $\{(T,\ S).$ $cdcl_{NOT}$*-restart* $S\ T \wedge$ $cdcl_{NOT}$*-inv* $(fst\ S)\}$ (**is** *wf ?A*)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-steps-bigger-than-bound*:
assumes
$cdcl_{NOT}$*-restart* $S\ T$ **and**
bound-inv A $(fst\ S)$ **and**
$cdcl_{NOT}$*-inv* $(fst\ S)$ **and**
$f\ (snd\ S) > \mu$*-bound* $A\ (fst\ S)$
shows *full1* $cdcl_{NOT}$ $(fst\ S)\ (fst\ T)$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-with-inv-inv-rtranclp-*$cdcl_{NOT}$:
assumes
inv: $cdcl_{NOT}$*-inv* S **and**
binv: *bound-inv* $A\ S$
shows $(\lambda S\ T.$ $cdcl_{NOT}$ $S\ T \wedge$ $cdcl_{NOT}$*-inv* $S \wedge$ *bound-inv* $A\ S)^{**}\ S\ T \longleftrightarrow$ $cdcl_{NOT}{}^{**}\ S\ T$
(**is** $?A^{**}\ S\ T \longleftrightarrow\ ?B^{**}\ S\ T$)
⟨*proof*⟩

lemma *no-step-*$cdcl_{NOT}$*-restart-no-step-*$cdcl_{NOT}$:
assumes
n-s: *no-step* $cdcl_{NOT}$*-restart* S **and**
inv: $cdcl_{NOT}$*-inv* $(fst\ S)$ **and**
binv: *bound-inv* $A\ (fst\ S)$
shows *no-step* $cdcl_{NOT}$ $(fst\ S)$
⟨*proof*⟩

end

5.2.6 Merging backjump and learning

locale $cdcl_{NOT}$*-merge-bj-learn-ops* =
  decide-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
  forget-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *forget-cond* +
  propagate-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *propagate-conds*
  for
    $trail :: 'st \Rightarrow ('v, unit)\ ann\text{-}lits$ **and**
    $clauses_{NOT} :: 'st \Rightarrow 'v\ clauses$ **and**
    $prepend\text{-}trail :: ('v, unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
    $tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
    $add\text{-}cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
    $remove\text{-}cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
    $propagate\text{-}conds :: ('v, unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow bool$ **and**
    $forget\text{-}cond :: 'v\ clause \Rightarrow 'st \Rightarrow bool$ +
  fixes $backjump\text{-}l\text{-}cond :: 'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
begin

We have a new backjump that combines the backjumping on the trail and the learning of the used clause (called C'' below)

inductive *backjump-l* **where**
backjump-l: $trail\ S = F'\ @\ Decided\ K\ \#\ F$
  $\Longrightarrow no\text{-}dup\ (trail\ S)$
  $\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ (add\text{-}cls_{NOT}\ C''\ S))$
  $\Longrightarrow C \in\#\ clauses_{NOT}\ S$
  $\Longrightarrow trail\ S \models as\ CNot\ C$
  $\Longrightarrow undefined\text{-}lit\ F\ L$
  $\Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S))$
  $\Longrightarrow clauses_{NOT}\ S \models pm\ C' + \{\#L\#\}$
  $\Longrightarrow C'' = C' + \{\#L\#\}$
  $\Longrightarrow F \models as\ CNot\ C'$
  $\Longrightarrow backjump\text{-}l\text{-}cond\ C\ C'\ L\ S\ T$
  $\Longrightarrow backjump\text{-}l\ S\ T$

Avoid (meaningless) simplification in the theorem generated by *inductive-cases*:

declare $reduce\text{-}trail\text{-}to_{NOT}$*-length-ne*[*simp del*] *Set.Un-iff*[*simp del*] *Set.insert-iff*[*simp del*]
inductive-cases *backjump-lE*: *backjump-l S T*
thm *backjump-lE*
declare $reduce\text{-}trail\text{-}to_{NOT}$*-length-ne*[*simp*] *Set.Un-iff*[*simp*] *Set.insert-iff*[*simp*]

inductive $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
$cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}decide_{NOT}$: $decide_{NOT}\ S\ S' \Longrightarrow cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ S'$ |
$cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}propagate_{NOT}$: $propagate_{NOT}\ S\ S' \Longrightarrow cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ S'$ |
$cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}backjump\text{-}l$: $backjump\text{-}l\ S\ S' \Longrightarrow cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ S'$ |
$cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}forget_{NOT}$: $forget_{NOT}\ S\ S' \Longrightarrow cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ S'$

lemma $cdcl_{NOT}$*-merged-bj-learn-no-dup-inv*:
  $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ T \Longrightarrow no\text{-}dup\ (trail\ S) \Longrightarrow no\text{-}dup\ (trail\ T)$
  ⟨*proof*⟩
end

locale $cdcl_{NOT}$*-merge-bj-learn-proxy* =
  $cdcl_{NOT}$*-merge-bj-learn-ops trail* $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
    propagate-conds forget-cond
    $\lambda C\ C'\ L'\ S\ T.\ backjump\text{-}l\text{-}cond\ C\ C'\ L'\ S\ T$
    $\wedge\ distinct\text{-}mset\ (C' + \{\#L'\#\}) \wedge \neg tautology\ (C' + \{\#L'\#\})$

for
  trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
  $clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
  prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
  tl-trail :: $'st \Rightarrow 'st$ **and**
  $add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  $remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  propagate-conds :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow bool$ **and**
  forget-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow bool$ **and**
  backjump-l-cond :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ +
fixes
  inv :: $'st \Rightarrow bool$
assumes
  bj-merge-can-jump:
  $\bigwedge S\ C\ F'\ K\ F\ L.$
    $inv\ S$
    $\Longrightarrow trail\ S = F'\ @\ Decided\ K\ \#\ F$
    $\Longrightarrow C \in\#\ clauses_{NOT}\ S$
    $\Longrightarrow trail\ S \models as\ CNot\ C$
    $\Longrightarrow undefined\text{-}lit\ F\ L$
    $\Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F))$
    $\Longrightarrow clauses_{NOT}\ S \models pm\ C' + \{\#L\#\}$
    $\Longrightarrow F \models as\ CNot\ C'$
    $\Longrightarrow \neg no\text{-}step\ backjump\text{-}l\ S$ **and**
  cdcl-merged-inv: $\bigwedge S\ T.\ cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
begin

abbreviation *backjump-conds* :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
  where
$backjump\text{-}conds \equiv \lambda C\ C'\ L'\ S\ T.\ distinct\text{-}mset\ (C' + \{\#L'\#\}) \wedge \neg tautology\ (C' + \{\#L'\#\})$

Without additional knowledge on *backjump-l-cond*, it is impossible to have the same invariant.

sublocale *dpll-with-backjumping-ops trail* $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  inv backjump-conds propagate-conds
⟨*proof*⟩

end

locale $cdcl_{NOT}$*-merge-bj-learn-proxy2* =
  $cdcl_{NOT}$*-merge-bj-learn-proxy trail* $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
    propagate-conds forget-cond backjump-l-cond inv
  for
  trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
  $clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
  prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
  tl-trail :: $'st \Rightarrow 'st$ **and**
  $add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  $remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  propagate-conds :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow bool$ **and**
  forget-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow bool$ **and**
  backjump-l-cond :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **and**
  inv :: $'st \Rightarrow bool$
begin

sublocale *conflict-driven-clause-learning-ops trail* $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$
  $remove\text{-}cls_{NOT}$ *inv backjump-conds propagate-conds*

λC -. *distinct-mset* $C \wedge \neg$*tautology* C
forget-cond
⟨*proof*⟩
end

locale *cdcl*$_{NOT}$*-merge-bj-learn* =
cdcl$_{NOT}$*-merge-bj-learn-proxy2 trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
propagate-conds forget-cond backjump-l-cond inv
for
trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
clauses$_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
backjump-l-cond :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow$ *bool* **and**
propagate-conds :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow$ *bool* **and**
forget-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow$ *bool* **and**
inv :: $'st \Rightarrow$ *bool* +
assumes
dpll-merge-bj-inv: $\bigwedge S\ T.$ *dpll-bj* $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *inv* T **and**
learn-inv: $\bigwedge S\ T.$ *learn* $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *inv* T
begin

sublocale
conflict-driven-clause-learning trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds
λC -. *distinct-mset* $C \wedge \neg$*tautology* C
forget-cond
⟨*proof*⟩

lemma *backjump-l-learn-backjump*:
assumes *bt*: *backjump-l* $S\ T$ **and** *inv*: *inv* S **and** *n-d*: *no-dup* (*trail* S)
shows $\exists\ C'\ L\ D.$ *learn* S (*add-cls*$_{NOT}$ $D\ S$)
$\wedge\ D = (C' + \{\#L\#\})$
$\wedge$ *backjump* (*add-cls*$_{NOT}$ $D\ S$) T
$\wedge$ *atms-of* $(C' + \{\#L\#\}) \subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S))
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-merged-bj-learn-is-tranclp-cdcl*$_{NOT}$:
cdcl$_{NOT}$*-merged-bj-learn* $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *cdcl*$_{NOT}{}^{++}$ $S\ T$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-is-rtranclp-cdcl*$_{NOT}$*-and-inv*:
cdcl$_{NOT}$*-merged-bj-learn*** $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *cdcl*$_{NOT}{}^{**}$ $S\ T \wedge$ *inv* T
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-is-rtranclp-cdcl*$_{NOT}$:
cdcl$_{NOT}$*-merged-bj-learn*** $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$*no-dup* (*trail* S) $\Longrightarrow$ *cdcl*$_{NOT}{}^{**}$ $S\ T$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-inv*:
cdcl$_{NOT}$*-merged-bj-learn*** $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *inv* T
⟨*proof*⟩

definition $\mu_C{}'$ :: $'v$ *clause set* $\Rightarrow 'st \Rightarrow$ *nat* **where**

$\mu_C{}'\ A\ T \equiv \mu_C\ (1{+}card\ (atms\text{-}of\text{-}ms\ A))\ (2{+}card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)$

definition $\mu_{CDCL}{}'$*-merged* :: *'v clause set* ⇒ *'st* ⇒ *nat* **where**
$\mu_{CDCL}{}'$*-merged A T* ≡
$((2{+}card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1{+}card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ T) * 2 + card\ (set\text{-}mset\ (clauses_{NOT}\ T))$

lemma $cdcl_{NOT}$*-decreasing-measure'*:
assumes
$cdcl_{NOT}$*-merged-bj-learn S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-trail: *atm-of ‘ lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows $\mu_{CDCL}{}'$*-merged A T* < $\mu_{CDCL}{}'$*-merged A S*
⟨*proof*⟩

lemma *wf-*$cdcl_{NOT}$*-merged-bj-learn*:
assumes
fin-A: *finite A*
shows *wf* {(*T*, *S*).
(*inv S* ∧ *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of ‘ lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*))
∧ $cdcl_{NOT}$*-merged-bj-learn S T*}
⟨*proof*⟩

lemma *tranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-tranclp*:
assumes
$cdcl_{NOT}$*-merged-bj-learn*$^{++}$ *S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-trail: *atm-of ‘ lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A[*simp*]: *finite A*
shows (*T*, *S*) ∈ {(*T*, *S*).
(*inv S* ∧ *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of ‘ lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*))
∧ $cdcl_{NOT}$*-merged-bj-learn S T*}$^+$ (**is** - ∈ *?P*$^+$)
⟨*proof*⟩

lemma *wf-tranclp-*$cdcl_{NOT}$*-merged-bj-learn*:
assumes *finite A*
shows *wf* {(*T*, *S*).
(*inv S* ∧ *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of ‘ lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*))
∧ $cdcl_{NOT}$*-merged-bj-learn*$^{++}$ *S T*}
⟨*proof*⟩

lemma *backjump-no-step-backjump-l*:
backjump S T ⟹ *inv S* ⟹ ¬*no-step backjump-l S*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-merged-bj-learn-final-state*:
fixes *A* :: *'v clause set* **and** *S T* :: *'st*
assumes

n-s: *no-step* $cdcl_{NOT}$*-merged-bj-learn* S **and**
atms-S: *atms-of-mm* ($clauses_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite A **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ S))
$\vee$ (*trail* S $\models$*asm* $clauses_{NOT}$ S $\wedge$ *satisfiable* (*set-mset* ($clauses_{NOT}$ S)))
⟨*proof*⟩

lemma *full-*$cdcl_{NOT}$*-merged-bj-learn-final-state*:
fixes A :: $'v$ *clause set* **and** S T :: $'st$
assumes
full: *full* $cdcl_{NOT}$*-merged-bj-learn* S T **and**
atms-S: *atms-of-mm* ($clauses_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite A **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ T))
$\vee$ (*trail* T $\models$*asm* $clauses_{NOT}$ T $\wedge$ *satisfiable* (*set-mset* ($clauses_{NOT}$ T)))
⟨*proof*⟩

end

5.2.7 Instantiations

In this section, we instantiate the previous locales to ensure that the assumption are not contradictory.

locale $cdcl_{NOT}$*-with-backtrack-and-restarts* =
conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt
trail $clauses_{NOT}$ *prepend-trail tl-trail add-*cls_{NOT} *remove-*cls_{NOT}
inv backjump-conds propagate-conds learn-restrictions forget-restrictions
for
trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
$clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-cls_{NOT} :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls_{NOT} :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
inv :: $'st \Rightarrow$ *bool* **and**
backjump-conds :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow$ *bool* **and**
propagate-conds :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow$ *bool* **and**
learn-restrictions forget-restrictions :: $'v$ *clause* $\Rightarrow 'st \Rightarrow$ *bool*
+
fixes f :: *nat* $\Rightarrow$ *nat*
assumes
unbounded: *unbounded* f **and** *f-ge-1*: $\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$ **and**
inv-restart: $\bigwedge S\ T.$ *inv* $S \Longrightarrow T \sim$ *reduce-trail-to*$_{NOT}$ ([]::$'a$ *list*) $S \Longrightarrow$ *inv* T
begin

lemma *bound-inv-inv*:
assumes

inv S **and**
n-d: *no-dup* (*trail S*) **and**
atms-clss-S-A: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail-S-A:*atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
finite A **and**
$cdcl_{NOT}$: $cdcl_{NOT}$ *S T*
shows
atms-of-mm ($clauses_{NOT}$ *T*) ⊆ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A* **and**
finite A
⟨*proof*⟩

sublocale $cdcl_{NOT}$*-increasing-restarts-ops* λ*S T*. *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S* $cdcl_{NOT}$ *f*
λ*A S*. *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧
finite A
μ_{CDCL}' λ*S*. *inv S* ∧ *no-dup* (*trail S*)
μ_{CDCL}'*-bound*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-with-restart-*μ_{CDCL}'*-le-*μ_{CDCL}'*-bound*:
assumes
$cdcl_{NOT}$: $cdcl_{NOT}$*-restart* (*T*, *a*) (*V*, *b*) **and**
$cdcl_{NOT}$*-inv*:
inv T
no-dup (*trail T*) **and**
bound-inv:
atms-of-mm ($clauses_{NOT}$ *T*) ⊆ *atms-of-ms A*
atm-of ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
finite A
shows μ_{CDCL}' *A V* ≤ μ_{CDCL}'*-bound A T*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-with-restart-*μ_{CDCL}'*-bound-le-*μ_{CDCL}'*-bound*:
assumes
$cdcl_{NOT}$: $cdcl_{NOT}$*-restart* (*T*, *a*) (*V*, *b*) **and**
$cdcl_{NOT}$*-inv*:
inv T
no-dup (*trail T*) **and**
bound-inv:
atms-of-mm ($clauses_{NOT}$ *T*) ⊆ *atms-of-ms A*
atm-of ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
finite A
shows μ_{CDCL}'*-bound A V* ≤ μ_{CDCL}'*-bound A T*
⟨*proof*⟩

sublocale $cdcl_{NOT}$*-increasing-restarts* - - - - - -
f
λ*S T*. *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S*
λ*A S*. *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A*
∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧ *finite A*
μ_{CDCL}' $cdcl_{NOT}$
λ*S*. *inv S* ∧ *no-dup* (*trail S*)
μ_{CDCL}'*-bound*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-all-decomposition-implies*:

assumes $cdcl_{NOT}$*-restart* S T **and**
inv (*fst* S) **and**
no-dup (*trail* (*fst* S))
all-decomposition-implies-m ($clauses_{NOT}$ (*fst* S)) (*get-all-ann-decomposition* (*trail* (*fst* S)))
shows
all-decomposition-implies-m ($clauses_{NOT}$ (*fst* T)) (*get-all-ann-decomposition* (*trail* (*fst* T)))
⟨*proof*⟩

lemma *rtranclp*-$cdcl_{NOT}$*-restart-all-decomposition-implies*:
assumes $cdcl_{NOT}$*-restart*** S T **and**
inv: *inv* (*fst* S) **and**
n-d: *no-dup* (*trail* (*fst* S)) **and**
decomp:
all-decomposition-implies-m ($clauses_{NOT}$ (*fst* S)) (*get-all-ann-decomposition* (*trail* (*fst* S)))
shows
all-decomposition-implies-m ($clauses_{NOT}$ (*fst* T)) (*get-all-ann-decomposition* (*trail* (*fst* T)))
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-sat-ext-iff*:
assumes
st: $cdcl_{NOT}$*-restart* S T **and**
n-d: *no-dup* (*trail* (*fst* S)) **and**
inv: *inv* (*fst* S)
shows I ⊨*sextm* $clauses_{NOT}$ (*fst* S) ⟷ I ⊨*sextm* $clauses_{NOT}$ (*fst* T)
⟨*proof*⟩

lemma *rtranclp*-$cdcl_{NOT}$*-restart-sat-ext-iff*:
fixes S T :: *'st* × *nat*
assumes
st: $cdcl_{NOT}$*-restart*** S T **and**
n-d: *no-dup* (*trail* (*fst* S)) **and**
inv: *inv* (*fst* S)
shows I ⊨*sextm* $clauses_{NOT}$ (*fst* S) ⟷ I ⊨*sextm* $clauses_{NOT}$ (*fst* T)
⟨*proof*⟩

theorem *full*-$cdcl_{NOT}$*-restart-backjump-final-state*:
fixes A :: *'v clause set* **and** S T :: *'st*
assumes
full: *full* $cdcl_{NOT}$*-restart* (S, n) (T, m) **and**
atms-S: *atms-of-mm* ($clauses_{NOT}$ S) ⊆ *atms-of-ms* A **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail* S) ⊆ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
fin-A[*simp*]: *finite* A **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ S))
∨ (*lits-of-l* (*trail* T) ⊨*sextm* $clauses_{NOT}$ S ∧ *satisfiable* (*set-mset* ($clauses_{NOT}$ S)))
⟨*proof*⟩
end — end of $cdcl_{NOT}$*-with-backtrack-and-restarts* locale

The restart does only reset the trail, contrary to Weidenbach's version where forget and restart are always combined. But there is a forget rule.

locale $cdcl_{NOT}$*-merge-bj-learn-with-backtrack-restarts* =
$cdcl_{NOT}$*-merge-bj-learn* *trail* $clauses_{NOT}$ *prepend-trail* *tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
$\lambda C\ C'\ L'\ S\ T.$ *distinct-mset* (C' + {#L'#}) ∧ *backjump-l-cond* C C' L' S T
propagate-conds *forget-conds* *inv*

for
  trail :: $'st \Rightarrow ('v,\ unit)$ *ann-lits* **and**
  $clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
  prepend-trail :: $('v,\ unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
  tl-trail :: $'st \Rightarrow 'st$ **and**
  $add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  $remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
  propagate-conds :: $('v,\ unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow bool$ **and**
  inv :: $'st \Rightarrow bool$ **and**
  forget-conds :: $'v$ *clause* $\Rightarrow 'st \Rightarrow bool$ **and**
  backjump-l-cond :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
  $+$
fixes $f :: nat \Rightarrow nat$
assumes
  unbounded: *unbounded f* **and** *f-ge-1*: $\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$ **and**
  inv-restart:$\bigwedge S\ T.\ inv\ S \Longrightarrow T \sim reduce\text{-}trail\text{-}to_{NOT}\ []\ S \Longrightarrow inv\ T$
begin

definition *not-simplified-cls* :: $'b$ *literal multiset multiset* $\Rightarrow 'b$ *literal multiset multiset*
where
not-simplified-cls $A \equiv \{\#C \in\#\ A.\ C \notin simple\text{-}clss\ (atms\text{-}of\text{-}mm\ A)\#\}$

lemma *not-simplified-cls-tautology-distinct-mset*:
  not-simplified-cls $A = \{\#C \in\#\ A.\ tautology\ C \vee \neg distinct\text{-}mset\ C\#\}$
  ⟨*proof*⟩

lemma *simple-clss-or-not-simplified-cls*:
  assumes *atms-of-mm* $(clauses_{NOT}\ S) \subseteq$ *atms-of-ms* A **and**
    $x \in\#\ clauses_{NOT}\ S$ **and** *finite A*
  shows $x \in simple\text{-}clss\ (atms\text{-}of\text{-}ms\ A) \vee x \in\#\ not\text{-}simplified\text{-}cls\ (clauses_{NOT}\ S)$
⟨*proof*⟩

lemma $cdcl_{NOT}$*-merged-bj-learn-clauses-bound*:
  assumes
    $cdcl_{NOT}$*-merged-bj-learn* $S\ T$ **and**
    inv: *inv* S **and**
    atms-clss: *atms-of-mm* $(clauses_{NOT}\ S) \subseteq$ *atms-of-ms* A **and**
    atms-trail: *atm-of* ‘(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
    n-d: *no-dup* (*trail* S) **and**
    fin-A[*simp*]: *finite* A
  shows *set-mset* $(clauses_{NOT}\ T) \subseteq$ *set-mset* (*not-simplified-cls* $(clauses_{NOT}\ S)$)
    $\cup$ *simple-clss* (*atms-of-ms* A)
  ⟨*proof*⟩

lemma $cdcl_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
  assumes $cdcl_{NOT}$*-merged-bj-learn* $S\ T$
  shows *not-simplified-cls* $(clauses_{NOT}\ T) \subseteq\#$ *not-simplified-cls* $(clauses_{NOT}\ S)$
  ⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
  assumes $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn^{**}\ S\ T$
  shows *not-simplified-cls* $(clauses_{NOT}\ T) \subseteq\#$ *not-simplified-cls* $(clauses_{NOT}\ S)$
  ⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-clauses-bound*:
  assumes

$cdcl_{NOT}$*-merged-bj-learn*** S T **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atm-of ‘(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite[*simp*]: *finite* A
shows *set-mset* (*clauses*$_{NOT}$ T) $\subseteq$ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ S))
$\cup$ *simple-clss* (*atms-of-ms* A)
⟨*proof*⟩

abbreviation $\mu_{CDCL}{}'$*-bound* **where**
$\mu_{CDCL}{}'$*-bound* A $T \equiv$ ((*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))) $*$ *2*
\+ *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ T)))
\+ *3* ^ *card* (*atms-of-ms* A)

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound-card*:
assumes
$cdcl_{NOT}$*-merged-bj-learn*** S T **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atm-of ‘(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite: *finite* A
shows $\mu_{CDCL}{}'$*-merged* A $T \leq \mu_{CDCL}{}'$*-bound* A S
⟨*proof*⟩

sublocale $cdcl_{NOT}$*-increasing-restarts-ops* λS T. $T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) S
$cdcl_{NOT}$*-merged-bj-learn* f
λA S. *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A
$\wedge$ *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A $\wedge$ *finite* A
$\mu_{CDCL}{}'$*-merged*
λS. *inv* S $\wedge$ *no-dup* (*trail* S)
$\mu_{CDCL}{}'$*-bound*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-*$\mu_{CDCL}{}'$*-merged-le-*$\mu_{CDCL}{}'$*-bound*:
assumes
$cdcl_{NOT}$*-restart* T V
inv (*fst* T) **and**
no-dup (*trail* (*fst* T)) **and**
atms-of-mm (*clauses*$_{NOT}$ (*fst* T)) $\subseteq$ *atms-of-ms* A **and**
atm-of ‘ *lits-of-l* (*trail* (*fst* T)) $\subseteq$ *atms-of-ms* A **and**
finite A
shows $\mu_{CDCL}{}'$*-merged* A (*fst* V) $\leq \mu_{CDCL}{}'$*-bound* A (*fst* T)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-*$\mu_{CDCL}{}'$*-bound-le-*$\mu_{CDCL}{}'$*-bound*:
assumes
$cdcl_{NOT}$*-restart* T V **and**
no-dup (*trail* (*fst* T)) **and**
inv (*fst* T) **and**
fin: *finite* A
shows $\mu_{CDCL}{}'$*-bound* A (*fst* V) $\leq \mu_{CDCL}{}'$*-bound* A (*fst* T)
⟨*proof*⟩

sublocale *cdcl*$_{NOT}$*-increasing-restarts* - - - - - - *f*
$\lambda S\ T.\ T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S*
$\lambda A\ S.$ *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* $\wedge$ *finite A*
μ_{CDCL}'*-merged cdcl*$_{NOT}$*-merged-bj-learn*
$\lambda S.$ *inv S* $\wedge$ *no-dup* (*trail S*)
$\lambda A\ T.$ ((*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))) $*$ *2*
\+ *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ *T*)))
\+ *3* ^ *card* (*atms-of-ms A*)
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-restart-eq-sat-iff*:
assumes
cdcl$_{NOT}$*-restart S T* **and**
no-dup (*trail* (*fst S*))
inv (*fst S*)
shows *I*$\models$*sextm clauses*$_{NOT}$ (*fst S*) $\longleftrightarrow$ *I* $\models$*sextm clauses*$_{NOT}$ (*fst T*)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-restart-eq-sat-iff*:
assumes
cdcl$_{NOT}$*-restart*** *S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*))
shows *I*$\models$*sextm clauses*$_{NOT}$ (*fst S*) $\longleftrightarrow$ *I* $\models$*sextm clauses*$_{NOT}$ (*fst T*)
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-restart-all-decomposition-implies-m*:
assumes
cdcl$_{NOT}$*-restart S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst S*))
(*get-all-ann-decomposition* (*trail* (*fst S*)))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst T*))
(*get-all-ann-decomposition* (*trail* (*fst T*)))
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-restart-all-decomposition-implies-m*:
assumes
cdcl$_{NOT}$*-restart*** *S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst S*))
(*get-all-ann-decomposition* (*trail* (*fst S*)))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst T*))
(*get-all-ann-decomposition* (*trail* (*fst T*)))
⟨*proof*⟩

lemma *full-cdcl*$_{NOT}$*-restart-normal-form*:
assumes
full: *full cdcl*$_{NOT}$*-restart S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst S*))
(*get-all-ann-decomposition* (*trail* (*fst S*))) **and**
atms-cls: *atms-of-mm* (*clauses*$_{NOT}$ (*fst S*)) $\subseteq$ *atms-of-ms A* **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail* (*fst S*)) $\subseteq$ *atms-of-ms A* **and**
fin: *finite A*
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst S*)))

$\vee$ *lits-of-l* (*trail* (*fst* T)) $\models$*sextm* $clauses_{NOT}$ (*fst* S) $\wedge$
satisfiable (*set-mset* ($clauses_{NOT}$ (*fst* S)))
⟨*proof*⟩

corollary $full\text{-}cdcl_{NOT}$*-restart-normal-form-init-state*:
assumes
init-state: *trail* S = [] $clauses_{NOT}$ S = N **and**
full: *full* $cdcl_{NOT}$*-restart* (S, *0*) T **and**
inv: *inv* S
shows *unsatisfiable* (*set-mset* N)
$\vee$ *lits-of-l* (*trail* (*fst* T)) $\models$*sextm* N $\wedge$ *satisfiable* (*set-mset* N)
⟨*proof*⟩

end

end
theory *DPLL-NOT*
imports *CDCL-NOT*
begin

5.3 DPLL as an instance of NOT

5.3.1 DPLL with simple backtrack

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

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

inductive-cases *backtrackE*[*elim*]: *backtrack* (M, N) (M', N')
lemma *backtrack-is-backjump*:
fixes M M' :: ($'v$, *unit*) *ann-lits*
assumes
backtrack: *backtrack* (M, N) (M', N') **and**
no-dup: (*no-dup* $\circ$ *fst*) (M, N) **and**
decomp: *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
shows
$\exists$ C F' K F L l C'.
M = F' @ *Decided* K # F $\wedge$
M' = *Propagated* L l # F $\wedge$ N = N' $\wedge$ C $\in$# N $\wedge$ F' @ *Decided* K # F $\models$*as* *CNot* C $\wedge$
undefined-lit F L $\wedge$ *atm-of* L $\in$ *atms-of-mm* N $\cup$ *atm-of* ‘ *lits-of-l* (F' @ *Decided* K # F) $\wedge$
N $\models$*pm* C' + {#L#} $\wedge$ F $\models$*as* *CNot* C'
⟨*proof*⟩

lemma *backtrack-is-backjump*$'$:
fixes M M' :: ($'v$, *unit*) *ann-lits*
assumes
backtrack: *backtrack* S T **and**
no-dup: (*no-dup* $\circ$ *fst*) S **and**
decomp: *all-decomposition-implies-m* (*snd* S) (*get-all-ann-decomposition* (*fst* S))
shows
$\exists$ C F' K F L l C'.

fst S = *F′* @ *Decided K* # *F* ∧
T = (*Propagated L l* # *F*, *snd S*) ∧ *C* ∈# *snd S* ∧ *fst S* ⊨*as CNot C*
∧ *undefined-lit F L* ∧ *atm-of L* ∈ *atms-of-mm* (*snd S*) ∪ *atm-of* ‘ *lits-of-l* (*fst S*) ∧
snd S ⊨*pm C′* + {#*L*#} ∧ *F* ⊨*as CNot C′*
⟨*proof*⟩

sublocale *dpll-state*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*) λ(*M*, *N*). (*tl M*, *N*)
λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*)
⟨*proof*⟩

sublocale *backjumping-ops*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*) λ(*M*, *N*). (*tl M*, *N*)
λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*) λ- - - *S T*. *backtrack S T*
⟨*proof*⟩
thm *reduce-trail-to*$_{NOT}$*-clauses*

lemma *reduce-trail-to*$_{NOT}$:
reduce-trail-to$_{NOT}$ *F S* =
(*if length* (*fst S*) ≥ *length F*
then drop (*length* (*fst S*) − *length F*) (*fst S*)
else [],
snd S) (**is** *?R* = *?C*)
⟨*proof*⟩

lemma *backtrack-is-backjump′′*:
fixes *M M′* :: (*′v*, *unit*) *ann-lits*
assumes
backtrack: *backtrack S T* **and**
no-dup: (*no-dup* ∘ *fst*) *S* **and**
decomp: *all-decomposition-implies-m* (*snd S*) (*get-all-ann-decomposition* (*fst S*))
shows *backjump S T*
⟨*proof*⟩

lemma *can-do-bt-step*:
assumes
M: *fst S* = *F′* @ *Decided K* # *F* **and**
C ∈# *snd S* **and**
C: *fst S* ⊨*as CNot C*
shows ¬ *no-step backtrack S*
⟨*proof*⟩

end

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping-ops*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*)
λ(*M*, *N*). (*tl M*, *N*) λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*)
λ(*M*, *N*). *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)
λ- - - *S T*. *backtrack S T*
λ- -. *True*
⟨*proof*⟩

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*)
λ(*M*, *N*). (*tl M*, *N*) λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*)
λ(*M*, *N*). *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)

λ- - - S T. *backtrack* S T
λ- -. *True*
⟨*proof*⟩

context *dpll-with-backtrack*
begin
lemma *wf-tranclp-dpll-inital-state*:
assumes *fin*: *finite A*
shows *wf* $\{((M'::('v,\ unit)\ ann\text{-}lits,\ N'::'v\ clauses),\ ([],\ N))|M'\ N'\ N.$
$dpll\text{-}bj^{++}\ ([],\ N)\ (M',\ N') \wedge atms\text{-}of\text{-}mm\ N \subseteq atms\text{-}of\text{-}ms\ A\}$
⟨*proof*⟩

corollary *full-dpll-final-state-conclusive*:
fixes M M' :: ('*v*, *unit*) *ann-lits*
assumes
full: *full dpll-bj* ([], N) (M', N')
shows *unsatisfiable* (*set-mset* N) $\vee$ (M' $\models$*asm* N $\wedge$ *satisfiable* (*set-mset* N))
⟨*proof*⟩

corollary *full-dpll-normal-form-from-init-state*:
fixes M M' :: ('*v*, *unit*) *ann-lits*
assumes
full: *full dpll-bj* ([], N) (M', N')
shows M' $\models$*asm* N $\longleftrightarrow$ *satisfiable* (*set-mset* N)
⟨*proof*⟩

interpretation *conflict-driven-clause-learning-ops*
fst snd λL (M, N). (L # M, N)
λ(M, N). (*tl* M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N)
λ(M, N). *no-dup* M $\wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True* λ- -. *False* λ- -. *False*
⟨*proof*⟩

interpretation *conflict-driven-clause-learning*
fst snd λL (M, N). (L # M, N)
λ(M, N). (*tl* M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N)
λ(M, N). *no-dup* M $\wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True* λ- -. *False* λ- -. *False*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-is-dpll*:
$cdcl_{NOT}$ S T $\longleftrightarrow$ *dpll-bj* S T
⟨*proof*⟩

Another proof of termination:

lemma *wf* {(T, S). *dpll-bj* S T $\wedge$ $cdcl_{NOT}$*-NOT-all-inv* A S}
⟨*proof*⟩
end

5.3.2 Adding restarts

This was mainly a test whether it was possible to instantiate the assumption of the locale.

locale *dpll-withbacktrack-and-restarts* =

dpll-with-backtrack +
fixes f :: *nat* $\Rightarrow$ *nat*
assumes *unbounded*: *unbounded f* **and** *f-ge-1*:$\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$
begin
sublocale $cdcl_{NOT}$*-increasing-restarts*
fst snd $\lambda L\ (M,\ N).\ (L\ \#\ M,\ N)\ \lambda(M,\ N).\ (tl\ M,\ N)$
$\lambda C\ (M,\ N).\ (M,\ \{\#C\#\} + N)\ \lambda C\ (M,\ N).\ (M,\ removeAll\text{-}mset\ C\ N)\ f\ \lambda(\text{-},\ N)\ S.\ S = ([],\ N)$
$\lambda A\ (M,\ N).$ *atms-of-mm* $N \subseteq$ *atms-of-ms* $A\ \wedge$ *atm-of* ' *lits-of-l* $M \subseteq$ *atms-of-ms* $A\ \wedge$ *finite* A
$\wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
$\lambda A\ T.\ (2+card\ (atms\text{-}of\text{-}ms\ A))\ \widehat{}\ (1+card\ (atms\text{-}of\text{-}ms\ A))$
$-\ \mu_C\ (1+card\ (atms\text{-}of\text{-}ms\ A))\ (2+card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)$ *dpll-bj*
$\lambda(M,\ N).$ *no-dup* $M\ \wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
$\lambda A\ \text{-}.\ (2+card\ (atms\text{-}of\text{-}ms\ A))\ \widehat{}\ (1+card\ (atms\text{-}of\text{-}ms\ A))$
⟨*proof*⟩
end

end
theory *DPLL-W*
imports *Main Partial-Clausal-Logic Partial-Annotated-Clausal-Logic List-More Wellfounded-More DPLL-NOT*
begin

5.4 Weidenbach's DPLL

5.4.1 Rules

type-synonym $'a\ dpll_W$*-ann-lit* = $('a,\ unit)$ *ann-lit*
type-synonym $'a\ dpll_W$*-ann-lits* = $('a,\ unit)$ *ann-lits*
type-synonym $'v\ dpll_W$*-state* = $'v\ dpll_W$*-ann-lits* $\times\ 'v$ *clauses*

abbreviation *trail* :: $'v\ dpll_W$*-state* $\Rightarrow\ 'v\ dpll_W$*-ann-lits* **where**
trail $\equiv$ *fst*
abbreviation *clauses* :: $'v\ dpll_W$*-state* $\Rightarrow\ 'v$ *clauses* **where**
clauses $\equiv$ *snd*

inductive $dpll_W$ :: $'v\ dpll_W$*-state* $\Rightarrow\ 'v\ dpll_W$*-state* $\Rightarrow$ *bool* **where**
propagate: $C + \{\#L\#\} \in\#$ *clauses* $S \Longrightarrow$ *trail* $S \models as$ *CNot* $C \Longrightarrow$ *undefined-lit* (*trail* S) L
$\Longrightarrow dpll_W\ S$ (*Propagated* L () # *trail* S, *clauses* S) |
decided: *undefined-lit* (*trail* S) $L \Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* (*clauses* S)
$\Longrightarrow dpll_W\ S$ (*Decided* L # *trail* S, *clauses* S) |
backtrack: *backtrack-split* (*trail* S) = $(M',\ L\ \#\ M) \Longrightarrow$ *is-decided* $L \Longrightarrow D \in\#$ *clauses* S
$\Longrightarrow$ *trail* $S \models as$ *CNot* $D \Longrightarrow dpll_W\ S$ (*Propagated* $(-\ (lit\text{-}of\ L))$ () # M, *clauses* S)

5.4.2 Invariants

lemma $dpll_W$*-distinct-inv*:
assumes $dpll_W\ S\ S'$
and *no-dup* (*trail* S)
shows *no-dup* (*trail* S')
⟨*proof*⟩

lemma $dpll_W$*-consistent-interp-inv*:
assumes $dpll_W\ S\ S'$
and *consistent-interp* (*lits-of-l* (*trail* S))
and *no-dup* (*trail* S)
shows *consistent-interp* (*lits-of-l* (*trail* S'))

⟨*proof*⟩

lemma $dpll_W$*-vars-in-snd-inv*:
assumes $dpll_W$ *S S'*
and *atm-of* ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*clauses S*)
shows *atm-of* ‘ (*lits-of-l* (*trail S'*)) ⊆ *atms-of-mm* (*clauses S'*)
⟨*proof*⟩

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms* ((λ*a*. {#*lit-of a*#}) ‘ *c*) = *atm-of* ‘ *lit-of* ‘ *c*
⟨*proof*⟩

theorem 2.8.2 page 73 of Weidenbach's book

lemma $dpll_W$*-propagate-is-conclusion*:
assumes $dpll_W$ *S S'*
and *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
shows *all-decomposition-implies-m* (*clauses S'*) (*get-all-ann-decomposition* (*trail S'*))
⟨*proof*⟩

theorem 2.8.3 page 73 of Weidenbach's book

theorem $dpll_W$*-propagate-is-conclusion-of-decided*:
assumes $dpll_W$ *S S'*
and *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
shows *set-mset* (*clauses S'*) ∪ {{#*lit-of L*#} |*L*. *is-decided L* ∧ *L* ∈ *set* (*trail S'*)}
⊨*ps* (λ*a*. {#*lit-of a*#}) ‘ ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition* (*trail S'*)))
⟨*proof*⟩

theorem 2.8.4 page 73 of Weidenbach's book

lemma *only-propagated-vars-unsat*:
assumes *decided*: ∀ *x* ∈ *set M*. ¬ *is-decided x*
and *DN*: *D* ∈ *N* **and** *D*: *M* ⊨*as CNot D*
and *inv*: *all-decomposition-implies N* (*get-all-ann-decomposition M*)
and *atm-incl*: *atm-of* ‘ *lits-of-l M* ⊆ *atms-of-ms N*
shows *unsatisfiable N*
⟨*proof*⟩

lemma $dpll_W$*-same-clauses*:
assumes $dpll_W$ *S S'*
shows *clauses S* = *clauses S'*
⟨*proof*⟩

lemma *rtranclp-*$dpll_W$*-inv*:
assumes *rtranclp* $dpll_W$ *S S'*
and *inv*: *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-incl*: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
and *consistent-interp* (*lits-of-l* (*trail S*))
and *no-dup* (*trail S*)
shows *all-decomposition-implies-m* (*clauses S'*) (*get-all-ann-decomposition* (*trail S'*))
and *atm-of* ‘ *lits-of-l* (*trail S'*) ⊆ *atms-of-mm* (*clauses S'*)
and *clauses S* = *clauses S'*
and *consistent-interp* (*lits-of-l* (*trail S'*))
and *no-dup* (*trail S'*)
⟨*proof*⟩

definition *dpll*$_W$*-all-inv S* ≡
(*all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
∧ *consistent-interp* (*lits-of-l* (*trail S*))
∧ *no-dup* (*trail S*))

lemma *dpll*$_W$*-all-inv-dest*[*dest*]:
assumes *dpll*$_W$*-all-inv S*
shows *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
and *consistent-interp* (*lits-of-l* (*trail S*)) ∧ *no-dup* (*trail S*)
⟨*proof*⟩

lemma *rtranclp-dpll*$_W$*-all-inv*:
assumes *rtranclp dpll*$_W$ *S S'*
and *dpll*$_W$*-all-inv S*
shows *dpll*$_W$*-all-inv S'*
⟨*proof*⟩

lemma *dpll*$_W$*-all-inv*:
assumes *dpll*$_W$ *S S'*
and *dpll*$_W$*-all-inv S*
shows *dpll*$_W$*-all-inv S'*
⟨*proof*⟩

lemma *rtranclp-dpll*$_W$*-inv-starting-from-0*:
assumes *rtranclp dpll*$_W$ *S S'*
and *inv*: *trail S* = []
shows *dpll*$_W$*-all-inv S'*
⟨*proof*⟩

lemma *dpll*$_W$*-can-do-step*:
assumes *consistent-interp* (*set M*)
and *distinct M*
and *atm-of* ‘ (*set M*) ⊆ *atms-of-mm N*
shows *rtranclp dpll*$_W$ ([], *N*) (*map Decided M*, *N*)
⟨*proof*⟩

definition *conclusive-dpll*$_W$*-state* (*S*:: *'v dpll*$_W$*-state*) ⟷
(*trail S* ⊨*asm clauses S* ∨ ((∀ *L* ∈ *set* (*trail S*). ¬*is-decided L*)
∧ (∃ *C* ∈# *clauses S*. *trail S* ⊨*as CNot C*)))

theorem 2.8.6 page 74 of Weidenbach's book

lemma *dpll*$_W$*-strong-completeness*:
assumes *set M* ⊨*sm N*
and *consistent-interp* (*set M*)
and *distinct M*
and *atm-of* ‘ (*set M*) ⊆ *atms-of-mm N*
shows *dpll*$_W$** ([], *N*) (*map Decided M*, *N*)
and *conclusive-dpll*$_W$*-state* (*map Decided M*, *N*)
⟨*proof*⟩

theorem 2.8.5 page 73 of Weidenbach's book

lemma *dpll*$_W$*-sound*:
assumes
rtranclp dpll$_W$ ([], *N*) (*M*, *N*) **and**

$\forall S. \neg dpll_W\ (M, N)\ S$
shows $M \models asm\ N \longleftrightarrow satisfiable\ (set\text{-}mset\ N)$ (**is** $?A \longleftrightarrow ?B$)
⟨*proof*⟩

5.4.3 Termination

definition $dpll_W\text{-}mes\ M\ n =$
$map\ (\lambda l.\ if\ is\text{-}decided\ l\ then\ 2\ else\ (1{::}nat))\ (rev\ M)\ @\ replicate\ (n - length\ M)\ 3$

lemma $length\text{-}dpll_W\text{-}mes$:
assumes $length\ M \leq n$
shows $length\ (dpll_W\text{-}mes\ M\ n) = n$
⟨*proof*⟩

lemma *distinctcard-atm-of-lit-of-eq-length*:
assumes *no-dup S*
shows $card\ (atm\text{-}of\ `\ lits\text{-}of\text{-}l\ S) = length\ S$
⟨*proof*⟩

lemma $dpll_W\text{-}card\text{-}decrease$:
assumes *dpll*: $dpll_W\ S\ S'$ **and** $length\ (trail\ S') \leq card\ vars$
and $length\ (trail\ S) \leq card\ vars$
shows $(dpll_W\text{-}mes\ (trail\ S')\ (card\ vars),\ dpll_W\text{-}mes\ (trail\ S)\ (card\ vars))$
$\in lexn\ \{(a, b).\ a < b\}\ (card\ vars)$
⟨*proof*⟩

theorem 2.8.7 page 74 of Weidenbach's book

lemma $dpll_W\text{-}card\text{-}decrease'$:
assumes *dpll*: $dpll_W\ S\ S'$
and *atm-incl*: $atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}mm\ (clauses\ S)$
and *no-dup*: $no\text{-}dup\ (trail\ S)$
shows $(dpll_W\text{-}mes\ (trail\ S')\ (card\ (atms\text{-}of\text{-}mm\ (clauses\ S'))),$
$dpll_W\text{-}mes\ (trail\ S)\ (card\ (atms\text{-}of\text{-}mm\ (clauses\ S)))) \in lex\ \{(a, b).\ a < b\}$
⟨*proof*⟩

lemma *wf-lexn*: $wf\ (lexn\ \{(a, b).\ (a{::}nat) < b\}\ (card\ (atms\text{-}of\text{-}mm\ (clauses\ S))))$
⟨*proof*⟩

lemma $dpll_W\text{-}wf$:
$wf\ \{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}$
⟨*proof*⟩

lemma $dpll_W\text{-}tranclp\text{-}star\text{-}commute$:
$\{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}^+ = \{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge tranclp\ dpll_W\ S\ S'\}$
(**is** $?A = ?B$)
⟨*proof*⟩

lemma $dpll_W\text{-}wf\text{-}tranclp$: $wf\ \{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W{}^{++}\ S\ S'\}$
⟨*proof*⟩

lemma $dpll_W\text{-}wf\text{-}plus$:
$wf\ \{(S', ([], N))|\ S'.\ dpll_W{}^{++}\ ([], N)\ S'\}$ (**is** $wf\ ?P$)
⟨*proof*⟩

5.4.4 Final States

Proposition 2.8.1: final states are the normal forms of $dpll_W$

lemma $dpll_W$*-no-more-step-is-a-conclusive-state*:
  assumes $\forall S'. \neg dpll_W\ S\ S'$
  shows *conclusive-*$dpll_W$*-state* S
⟨*proof*⟩

lemma $dpll_W$*-conclusive-state-correct*:
  assumes ${dpll_W}^{**}$ $([], N)$ (M, N) **and** *conclusive-*$dpll_W$*-state* (M, N)
  shows M $\models$*asm* $N \longleftrightarrow$ *satisfiable* (*set-mset* N) (**is** $?A \longleftrightarrow ?B$)
⟨*proof*⟩

5.4.5 Link with NOT's DPLL

interpretation $dpll_W$*-*$_{NOT}$: *dpll-with-backtrack* ⟨*proof*⟩

declare $dpll_W$*-*$_{NOT}$*.state-simp*$_{NOT}$[*simp del*]
lemma *state-eq*$_{NOT}$*-iff-eq*[*iff*, *simp*]: $dpll_W$*-*$_{NOT}$*.state-eq*$_{NOT}$ S $T \longleftrightarrow S = T$
  ⟨*proof*⟩
lemma $dpll_W$*-*$dpll_W$*-bj*:
  assumes *inv*: $dpll_W$*-all-inv* S **and** *dpll*: $dpll_W$ S T
  shows $dpll_W$*-*$_{NOT}$*.dpll-bj* S T
  ⟨*proof*⟩

lemma $dpll_W$*-bj-dpll*:
  assumes *inv*: $dpll_W$*-all-inv* S **and** *dpll*: $dpll_W$*-*$_{NOT}$*.dpll-bj* S T
  shows $dpll_W$ S T
  ⟨*proof*⟩

lemma *rtranclp-*$dpll_W$*-rtranclp-*$dpll_W$*-*$_{NOT}$:
  assumes ${dpll_W}^{**}$ S T **and** $dpll_W$*-all-inv* S
  shows $dpll_W$*-*$_{NOT}$*.dpll-bj*** S T
  ⟨*proof*⟩

lemma *rtranclp-dpll-rtranclp-*$dpll_W$:
  assumes $dpll_W$*-*$_{NOT}$*.dpll-bj*** S T **and** $dpll_W$*-all-inv* S
  shows ${dpll_W}^{**}$ S T
  ⟨*proof*⟩

lemma *dpll-conclusive-state-correctness*:
  assumes $dpll_W$*-*$_{NOT}$*.dpll-bj*** $([], N)$ (M, N) **and** *conclusive-*$dpll_W$*-state* (M, N)
  shows M $\models$*asm* $N \longleftrightarrow$ *satisfiable* (*set-mset* N)
⟨*proof*⟩

end
theory *CDCL-W-Level*
imports *Partial-Annotated-Clausal-Logic*
begin

Level of literals and clauses

Getting the level of a variable, implies that the list has to be reversed. Here is the function after reversing.

abbreviation *count-decided* :: $('v, 'm)$ *ann-lits* $\Rightarrow$ *nat* **where**

count-decided l ≡ *length* (*filter is-decided l*)

abbreviation *get-level* :: ($'v$, $'m$) *ann-lits* ⇒ $'v$ *literal* ⇒ *nat* **where**
get-level S L ≡ *length* (*filter is-decided* (*dropWhile* (λ*S*. *atm-of* (*lit-of S*) ≠ *atm-of L*) *S*))

lemma *get-level-uminus*: *get-level M* (−*L*) = *get-level M L*
⟨*proof*⟩

lemma *atm-of-notin-get-rev-level-eq-0*[*simp*]:
assumes *atm-of L* ∉ *atm-of* ‘ *lits-of-l M*
shows *get-level M L* = *0*
⟨*proof*⟩

lemma *get-level-ge-0-atm-of-in*:
assumes *get-level M L* > *n*
shows *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*
⟨*proof*⟩

In *get-level* (resp. *get-level*), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

lemma *get-rev-level-skip*[*simp*]:
assumes *atm-of L* ∉ *atm-of* ‘ *lits-of-l M*
shows *get-level* (*M* @ *M'*) *L* = *get-level M' L*
⟨*proof*⟩

If the literal is at the beginning, then the end can be skipped

lemma *get-rev-level-skip-end*[*simp*]:
assumes *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*
shows *get-level* (*M* @ *M'*) *L* = *get-level M L* + *length* (*filter is-decided M'*)
⟨*proof*⟩

lemma *get-level-skip-beginning*:
assumes *atm-of L'* ≠ *atm-of* (*lit-of K*)
shows *get-level* (*K* # *M*) *L'* = *get-level M L'*
⟨*proof*⟩

lemma *get-level-skip-beginning-not-decided*[*simp*]:
assumes *atm-of L* ∉ *atm-of* ‘ *lits-of-l S*
and ∀ *s*∈*set S*. ¬*is-decided s*
shows *get-level* (*M* @ *S*) *L* = *get-level M L*
⟨*proof*⟩

lemma *get-level-skip-in-all-not-decided*:
fixes *M* :: ($'a$, $'b$) *ann-lits* **and** *L* :: $'a$ *literal*
assumes ∀ *m*∈*set M*. ¬ *is-decided m*
and *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*
shows *get-level M L* = *0*
⟨*proof*⟩

lemma *get-level-skip-all-not-decided*[*simp*]:
fixes *M*
assumes ∀ *m*∈*set M*. ¬ *is-decided m*
shows *get-level M L* = *0*
⟨*proof*⟩

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

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

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

lemma *get-maximum-level-ge-get-level*:
$L \in\# D \Longrightarrow get\text{-}maximum\text{-}level\ M\ D \geq get\text{-}level\ M\ L$
$\langle proof \rangle$

lemma *get-maximum-level-empty*[*simp*]:
$get\text{-}maximum\text{-}level\ M\ \{\#\} = 0$
$\langle proof \rangle$

lemma *get-maximum-level-exists-lit-of-max-level*:
$D \neq \{\#\} \Longrightarrow \exists L \in\# D.\ get\text{-}level\ M\ L = get\text{-}maximum\text{-}level\ M\ D$
$\langle proof \rangle$

lemma *get-maximum-level-empty-list*[*simp*]:
$get\text{-}maximum\text{-}level\ []\ D = 0$
$\langle proof \rangle$

lemma *get-maximum-level-single*[*simp*]:
$get\text{-}maximum\text{-}level\ M\ \{\#L\#\} = get\text{-}level\ M\ L$
$\langle proof \rangle$

lemma *get-maximum-level-plus*:
$get\text{-}maximum\text{-}level\ M\ (D + D') = max\ (get\text{-}maximum\text{-}level\ M\ D)\ (get\text{-}maximum\text{-}level\ M\ D')$
$\langle proof \rangle$

lemma *get-maximum-level-exists-lit*:
assumes n: $n > 0$
and max: $get\text{-}maximum\text{-}level\ M\ D = n$
shows $\exists L \in\# D.\ get\text{-}level\ M\ L = n$
$\langle proof \rangle$

lemma *get-maximum-level-skip-first*[*simp*]:
assumes $atm\text{-}of\ L \notin atms\text{-}of\ D$
shows $get\text{-}maximum\text{-}level\ (Propagated\ L\ C\ \#\ M)\ D = get\text{-}maximum\text{-}level\ M\ D$
$\langle proof \rangle$

lemma *get-maximum-level-skip-beginning*:
assumes DH: $\forall x \in atms\text{-}of\ D.\ x \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ c$
shows $get\text{-}maximum\text{-}level\ (c\ @\ H)\ D = get\text{-}maximum\text{-}level\ H\ D$
$\langle proof \rangle$

lemma *get-maximum-level-D-single-propagated*:
$get\text{-}maximum\text{-}level\ [Propagated\ x21\ x22]\ D = 0$
$\langle proof \rangle$

lemma *get-maximum-level-skip-un-decided-not-present*:
assumes
$\forall L \in\# D.\ atm\text{-}of\ L \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$ **and**
$\forall m \in set\ M.\ \neg\ is\text{-}decided\ m$
shows $get\text{-}maximum\text{-}level\ (M\ @\ aa)\ D = get\text{-}maximum\text{-}level\ aa\ D$

⟨proof⟩

lemma *get-maximum-level-union-mset*:
get-maximum-level M $(A \#\cup B)$ = *get-maximum-level M* $(A + B)$
⟨proof⟩

lemma *count-decided-rev*[*simp*]:
count-decided (*rev M*) = *count-decided M*
⟨proof⟩

lemma *count-decided-ge-get-level*[*simp*]:
count-decided M $\geq$ *get-level M L*
⟨proof⟩

lemma *count-decided-ge-get-maximum-level*:
count-decided M $\geq$ *get-maximum-level M D*
⟨proof⟩

fun *get-all-mark-of-propagated* **where**
get-all-mark-of-propagated [] = [] |
get-all-mark-of-propagated (*Decided - # L*) = *get-all-mark-of-propagated L* |
get-all-mark-of-propagated (*Propagated - mark # L*) = *mark # get-all-mark-of-propagated L*

lemma *get-all-mark-of-propagated-append*[*simp*]:
get-all-mark-of-propagated (*A @ B*) = *get-all-mark-of-propagated A @ get-all-mark-of-propagated B*
⟨proof⟩

Properties about the levels

lemma *atm-lit-of-set-lits-of-l*:
($\lambda l.$ *atm-of* (*lit-of l*)) ‘ *set xs* = *atm-of* ‘ *lits-of-l xs*
⟨proof⟩

lemma *le-count-decided-decomp*:
assumes *no-dup M*
shows*i* < *count-decided M* $\longleftrightarrow$ ($\exists$ *c K c'*. *M* = *c @ Decided K # c'* $\wedge$ *get-level M K* = *Suc i*)
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨proof⟩

end
theory *CDCL-W*
imports *List-More CDCL-W-Level Wellfounded-More Partial-Annotated-Clausal-Logic*

begin

Chapter 6

Weidenbach's CDCL

The organisation of the development is the following:

- `CDCL_W.thy` contains the specification of the rules: the rules and the strategy are defined, and we proof the correctness of CDCL.
- `CDCL_W_Termination.thy` contains the proof of termination.
- `CDCL_W_Merge.thy` contains a variant of the calculus: some rules of the raw calculus are always applied together (like the rules analysing the conflict and then backtracking). We define an equivalent version of the calculus where these rules are applied together. This is useful for implementations.
- `CDCL_WNOT.thy` proves the inclusion of Weidenbach's version of CDCL in NOT's version. We use here the version defined in `CDCL_W_Merge.thy`. We need this, because NOT's backjump corresponds to multiple applications of three rules in Weidenbach's calculus. We show also the termination of the calculus without strategy.

We have some variants build on the top of Weidenbach's CDCL calculus:

- `CDCL_W_Incremental.thy` adds incrementality on the top of `CDCL_W.thy`. The way we are doing it is not compatible with `CDCL_W_Merge.thy` , because we add conflicts and the `CDCL_W_Merge.thy` cannot analyse conflicts added externally, because the conflict and analyse are merged.
- `CDCL_W_Restart.thy` adds restart. It is built on the top of `CDCL_W_Merge.thy`.

6.1 Weidenbach's CDCL with Multisets

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

6.1.1 The State

We will abstract the representation of clause and clauses via two locales. We here use multisets, contrary to `CDCL_W_Abstract_State.thy` where we assume only the existence of a conversion to the state.

locale $state_W$*-ops* =

fixes
trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
init-clss :: *'st* ⇒ *'v clauses* **and**
learned-clss :: *'st* ⇒ *'v clauses* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
conflicting :: *'st* ⇒ *'v clause option* **and**

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

init-state :: *'v clauses* ⇒ *'st*
begin
abbreviation *hd-trail* :: *'st* ⇒ (*'v*, *'v clause*) *ann-lit* **where**
hd-trail S ≡ *hd* (*trail S*)

definition *clauses* :: *'st* ⇒ *'v clauses* **where**
clauses S = *init-clss S* + *learned-clss S*

abbreviation *resolve-cls* **where**
resolve-cls L D' E ≡ *remove1-mset* (−*L*) *D'* #∪ *remove1-mset L E*

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

We are using an abstract state to abstract away the detail of the implementation: we do not need to know how the clauses are represented internally, we just need to know that they can be converted to multisets.

Weidenbach state is a five-tuple composed of:

1. the trail is a list of decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause (*'v Partial-Clausal-Logic.clause*, standing for conflicting clause) and one for the initial and learned clauses (*'v Partial-Clausal-Logic.clause*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to *'v Partial-Clausal-Logic.clause* is enough (needed for function *hd-trail* below).
There are several axioms to state the independance of the different fields of the state: for example, adding a clause to the learned clauses does not change the trail.

locale $state_W$ =

$state_W$-*ops*

— functions about the state:

— getter:

trail init-clss learned-clss backtrack-lvl conflicting

— setter:

cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl update-conflicting

— Some specific states:

init-state

for

trail :: $'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lits$ **and**
init-clss :: $'st \Rightarrow 'v\ clauses$ **and**
learned-clss :: $'st \Rightarrow 'v\ clauses$ **and**
backtrack-lvl :: $'st \Rightarrow nat$ **and**
conflicting :: $'st \Rightarrow 'v\ clause\ option$ **and**

cons-trail :: $('v, 'v\ clause)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-learned-cls :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
update-backtrack-lvl :: $nat \Rightarrow 'st \Rightarrow 'st$ **and**
update-conflicting :: $'v\ clause\ option \Rightarrow 'st \Rightarrow 'st$ **and**

init-state :: $'v\ clauses \Rightarrow 'st$ +

assumes

cons-trail:
$\bigwedge S'.\ state\ st = (M, S') \Longrightarrow$
$state\ (cons\text{-}trail\ L\ st) = (L \# M, S')$ **and**

tl-trail:
$\bigwedge S'.\ state\ st = (M, S') \Longrightarrow state\ (tl\text{-}trail\ st) = (tl\ M, S')$ **and**

remove-cls:
$\bigwedge S'.\ state\ st = (M, N, U, S') \Longrightarrow$
$state\ (remove\text{-}cls\ C\ st) =$
$(M, removeAll\text{-}mset\ C\ N, removeAll\text{-}mset\ C\ U, S')$ **and**

add-learned-cls:
$\bigwedge S'.\ state\ st = (M, N, U, S') \Longrightarrow$
$state\ (add\text{-}learned\text{-}cls\ C\ st) = (M, N, \{\#C\#\} + U, S')$ **and**

update-backtrack-lvl:
$\bigwedge S'.\ state\ st = (M, N, U, k, S') \Longrightarrow$
$state\ (update\text{-}backtrack\text{-}lvl\ k'\ st) = (M, N, U, k', S')$ **and**

update-conflicting:
$state\ st = (M, N, U, k, D) \Longrightarrow$
$state\ (update\text{-}conflicting\ E\ st) = (M, N, U, k, E)$ **and**

init-state:
$state\ (init\text{-}state\ N) = ([], N, \{\#\}, 0, None)$

begin

lemma

trail-cons-trail[*simp*]:

trail (*cons-trail* L *st*) = L # *trail st* **and**
trail-tl-trail[*simp*]: *trail* (*tl-trail st*) = *tl* (*trail st*) **and**
trail-add-learned-cls[*simp*]:
trail (*add-learned-cls* C *st*) = *trail st* **and**
trail-remove-cls[*simp*]:
trail (*remove-cls* C *st*) = *trail st* **and**
trail-update-backtrack-lvl[*simp*]: *trail* (*update-backtrack-lvl k st*) = *trail st* **and**
trail-update-conflicting[*simp*]: *trail* (*update-conflicting E st*) = *trail st* **and**

init-clss-cons-trail[*simp*]:
init-clss (*cons-trail M st*) = *init-clss st*
and
init-clss-tl-trail[*simp*]:
init-clss (*tl-trail st*) = *init-clss st* **and**
init-clss-add-learned-cls[*simp*]:
init-clss (*add-learned-cls C st*) = *init-clss st* **and**
init-clss-remove-cls[*simp*]:
init-clss (*remove-cls C st*) = *removeAll-mset C* (*init-clss st*) **and**
init-clss-update-backtrack-lvl[*simp*]:
init-clss (*update-backtrack-lvl k st*) = *init-clss st* **and**
init-clss-update-conflicting[*simp*]:
init-clss (*update-conflicting E st*) = *init-clss st* **and**

learned-clss-cons-trail[*simp*]:
learned-clss (*cons-trail M st*) = *learned-clss st* **and**
learned-clss-tl-trail[*simp*]:
learned-clss (*tl-trail st*) = *learned-clss st* **and**
learned-clss-add-learned-cls[*simp*]:
learned-clss (*add-learned-cls C st*) = {#*C*#} + *learned-clss st* **and**
learned-clss-remove-cls[*simp*]:
learned-clss (*remove-cls C st*) = *removeAll-mset C* (*learned-clss st*) **and**
learned-clss-update-backtrack-lvl[*simp*]:
learned-clss (*update-backtrack-lvl k st*) = *learned-clss st* **and**
learned-clss-update-conflicting[*simp*]:
learned-clss (*update-conflicting E st*) = *learned-clss st* **and**

backtrack-lvl-cons-trail[*simp*]:
backtrack-lvl (*cons-trail M st*) = *backtrack-lvl st* **and**
backtrack-lvl-tl-trail[*simp*]:
backtrack-lvl (*tl-trail st*) = *backtrack-lvl st* **and**
backtrack-lvl-add-learned-cls[*simp*]:
backtrack-lvl (*add-learned-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-remove-cls[*simp*]:
backtrack-lvl (*remove-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-update-backtrack-lvl[*simp*]:
backtrack-lvl (*update-backtrack-lvl k st*) = *k* **and**
backtrack-lvl-update-conflicting[*simp*]:
backtrack-lvl (*update-conflicting E st*) = *backtrack-lvl st* **and**

conflicting-cons-trail[*simp*]:
conflicting (*cons-trail M st*) = *conflicting st* **and**
conflicting-tl-trail[*simp*]:
conflicting (*tl-trail st*) = *conflicting st* **and**
conflicting-add-learned-cls[*simp*]:
conflicting (*add-learned-cls C st*) = *conflicting st*
and

conflicting-remove-cls[*simp*]:
conflicting (*remove-cls* C *st*) = *conflicting st* **and**
conflicting-update-backtrack-lvl[*simp*]:
conflicting (*update-backtrack-lvl* k *st*) = *conflicting st* **and**
conflicting-update-conflicting[*simp*]:
conflicting (*update-conflicting* E *st*) = E **and**

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

⟨*proof*⟩

lemma
shows
clauses-cons-trail[*simp*]:
clauses (*cons-trail* M S) = *clauses* S **and**

clss-tl-trail[*simp*]: *clauses* (*tl-trail* S) = *clauses* S **and**
clauses-add-learned-cls-unfolded:
clauses (*add-learned-cls* U S) = {#U#} + *learned-clss* S + *init-clss* S
and
clauses-update-backtrack-lvl[*simp*]: *clauses* (*update-backtrack-lvl* k S) = *clauses* S **and**
clauses-update-conflicting[*simp*]: *clauses* (*update-conflicting* D S) = *clauses* S **and**
clauses-remove-cls[*simp*]:
clauses (*remove-cls* C S) = *removeAll-mset* C (*clauses* S) **and**
clauses-add-learned-cls[*simp*]:
clauses (*add-learned-cls* C S) = {#C#} + *clauses* S **and**
clauses-init-state[*simp*]: *clauses* (*init-state* N) = N
⟨*proof*⟩

abbreviation *incr-lvl* :: *'st* ⇒ *'st* **where**
incr-lvl S ≡ *update-backtrack-lvl* (*backtrack-lvl* S + *1*) S

definition *state-eq* :: *'st* ⇒ *'st* ⇒ *bool* (**infix** ∼ *50*) **where**
$S \sim T \longleftrightarrow$ *state* S = *state* T

lemma *state-eq-ref*[*simp*, *intro*]:
$S \sim S$
⟨*proof*⟩

lemma *state-eq-sym*:
$S \sim T \longleftrightarrow T \sim S$
⟨*proof*⟩

lemma *state-eq-trans*:
$S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$
⟨*proof*⟩

lemma
shows
state-eq-trail: $S \sim T \Longrightarrow$ *trail* S = *trail* T **and**
state-eq-init-clss: $S \sim T \Longrightarrow$ *init-clss* S = *init-clss* T **and**
state-eq-learned-clss: $S \sim T \Longrightarrow$ *learned-clss* S = *learned-clss* T **and**

state-eq-backtrack-lvl: $S \sim T \Longrightarrow$ *backtrack-lvl* S = *backtrack-lvl* T **and**
state-eq-conflicting: $S \sim T \Longrightarrow$ *conflicting* S = *conflicting* T **and**
state-eq-clauses: $S \sim T \Longrightarrow$ *clauses* S = *clauses* T **and**
state-eq-undefined-lit: $S \sim T \Longrightarrow$ *undefined-lit* (*trail* S) L = *undefined-lit* (*trail* T) L
⟨*proof*⟩

lemma *state-eq-conflicting-None*:
$S \sim T \Longrightarrow$ *conflicting* T = *None* $\Longrightarrow$ *conflicting* S = *None*
⟨*proof*⟩

We combine all simplification rules about *op* ∼ in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

lemmas *state-simp*[*simp*] = *state-eq-trail state-eq-init-clss state-eq-learned-clss state-eq-backtrack-lvl state-eq-conflicting state-eq-clauses state-eq-undefined-lit state-eq-conflicting-None*

function *reduce-trail-to* :: *'a list* ⇒ *'st* ⇒ *'st* **where**
reduce-trail-to F S =
(*if length* (*trail* S) = *length* F ∨ *trail* S = [] *then* S *else reduce-trail-to* F (*tl-trail* S))
⟨*proof*⟩
termination
⟨*proof*⟩

declare *reduce-trail-to.simps*[*simp del*]

lemma
shows
reduce-trail-to-Nil[*simp*]: *trail* S = [] $\Longrightarrow$ *reduce-trail-to* F S = S **and**
reduce-trail-to-eq-length[*simp*]: *length* (*trail* S) = *length* F $\Longrightarrow$ *reduce-trail-to* F S = S
⟨*proof*⟩

lemma *reduce-trail-to-length-ne*:
length (*trail* S) ≠ *length* F $\Longrightarrow$ *trail* S ≠ [] $\Longrightarrow$
reduce-trail-to F S = *reduce-trail-to* F (*tl-trail* S)
⟨*proof*⟩

lemma *trail-reduce-trail-to-length-le*:
assumes *length* F > *length* (*trail* S)
shows *trail* (*reduce-trail-to* F S) = []
⟨*proof*⟩

lemma *trail-reduce-trail-to-Nil*[*simp*]:
trail (*reduce-trail-to* [] S) = []
⟨*proof*⟩

lemma *clauses-reduce-trail-to-Nil*:
clauses (*reduce-trail-to* [] S) = *clauses* S
⟨*proof*⟩

lemma *reduce-trail-to-skip-beginning*:
assumes *trail* S = F' @ F
shows *trail* (*reduce-trail-to* F S) = F
⟨*proof*⟩

lemma *clauses-reduce-trail-to*[*simp*]:
clauses (*reduce-trail-to F S*) = *clauses S*
⟨*proof*⟩

lemma *conflicting-update-trail*[*simp*]:
conflicting (*reduce-trail-to F S*) = *conflicting S*
⟨*proof*⟩

lemma *backtrack-lvl-update-trail*[*simp*]:
backtrack-lvl (*reduce-trail-to F S*) = *backtrack-lvl S*
⟨*proof*⟩

lemma *init-clss-update-trail*[*simp*]:
init-clss (*reduce-trail-to F S*) = *init-clss S*
⟨*proof*⟩

lemma *learned-clss-update-trail*[*simp*]:
learned-clss (*reduce-trail-to F S*) = *learned-clss S*
⟨*proof*⟩

lemma *conflicting-reduce-trail-to*[*simp*]:
conflicting (*reduce-trail-to F S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
⟨*proof*⟩

lemma *trail-eq-reduce-trail-to-eq*:
trail S = *trail T* $\Longrightarrow$ *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
⟨*proof*⟩

lemma *reduce-trail-to-state-eq*$_{NOT}$*-compatible*:
assumes *ST*: *S* $\sim$ *T*
shows *reduce-trail-to F S* $\sim$ *reduce-trail-to F T*
⟨*proof*⟩

lemma *reduce-trail-to-trail-tl-trail-decomp*[*simp*]:
trail S = *F′* @ *Decided K* # *F* $\Longrightarrow$ (*trail* (*reduce-trail-to F S*)) = *F*
⟨*proof*⟩

lemma *reduce-trail-to-add-learned-cls*[*simp*]:
trail (*reduce-trail-to F* (*add-learned-cls C S*)) = *trail* (*reduce-trail-to F S*)
⟨*proof*⟩

lemma *reduce-trail-to-remove-learned-cls*[*simp*]:
trail (*reduce-trail-to F* (*remove-cls C S*)) = *trail* (*reduce-trail-to F S*)
⟨*proof*⟩

lemma *reduce-trail-to-update-conflicting*[*simp*]:
trail (*reduce-trail-to F* (*update-conflicting C S*)) = *trail* (*reduce-trail-to F S*)
⟨*proof*⟩

lemma *reduce-trail-to-update-backtrack-lvl*[*simp*]:
trail (*reduce-trail-to F* (*update-backtrack-lvl k S*)) = *trail* (*reduce-trail-to F S*)
⟨*proof*⟩

lemma *reduce-trail-to-length*:
length M = *length M′* $\Longrightarrow$ *reduce-trail-to M S* = *reduce-trail-to M′ S*
⟨*proof*⟩

lemma *trail-reduce-trail-to-drop*:
  trail (*reduce-trail-to F S*) =
    (*if length* (*trail S*) $\geq$ *length F*
    then drop (*length* (*trail S*) $-$ *length F*) (*trail S*)
    else [])
  ⟨*proof*⟩

lemma *in-get-all-ann-decomposition-trail-update-trail*[*simp*]:
  assumes *H*: (*L* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail S*))
  shows *trail* (*reduce-trail-to M1 S*) = *M1*
⟨*proof*⟩

lemma *conflicting-cons-trail-conflicting*[*simp*]:
  assumes *undefined-lit* (*trail S*) (*lit-of L*)
  shows
    conflicting (*cons-trail L S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
  ⟨*proof*⟩

lemma *conflicting-add-learned-cls-conflicting*[*simp*]:
  conflicting (*add-learned-cls C S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
  ⟨*proof*⟩

lemma *conflicting-update-backtracl-lvl*[*simp*]:
  conflicting (*update-backtrack-lvl k S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
  ⟨*proof*⟩

end — end of $state_W$ locale

6.1.2 CDCL Rules

Because of the strategy we will later use, we distinguish propagate, conflict from the other rules

locale $conflict\text{-}driven\text{-}clause\text{-}learning_W$ =
  $state_W$
    — functions for the state:
      — access functions:
    trail init-clss learned-clss backtrack-lvl conflicting
      — changing state:
    cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
    update-conflicting

      — get state:
    init-state
  for
    trail :: *'st* $\Rightarrow$ (*'v*, *'v clause*) *ann-lits* **and**
    init-clss :: *'st* $\Rightarrow$ *'v clauses* **and**
    learned-clss :: *'st* $\Rightarrow$ *'v clauses* **and**
    backtrack-lvl :: *'st* $\Rightarrow$ *nat* **and**
    conflicting :: *'st* $\Rightarrow$ *'v clause option* **and**

    cons-trail :: (*'v*, *'v clause*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    tl-trail :: *'st* $\Rightarrow$ *'st* **and**
    add-learned-cls :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    remove-cls :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    update-backtrack-lvl :: *nat* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**

update-conflicting :: *'v clause option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'v clauses* ⇒ *'st*
begin

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

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

inductive *conflict* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
conflict-rule:
conflicting S = None ⟹
D ∈# *clauses S* ⟹
trail S ⊨*as CNot D* ⟹
T ∼ *update-conflicting* (*Some D*) *S* ⟹
conflict S T

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

inductive *backtrack* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
backtrack-rule:
conflicting S = Some D ⟹
L ∈# *D* ⟹
(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) ⟹
get-level (*trail S*) *L* = *backtrack-lvl S* ⟹
get-level (*trail S*) *L* = *get-maximum-level* (*trail S*) *D* ⟹
get-maximum-level (*trail S*) (*D* − {#*L*#}) ≡ *i* ⟹
get-level (*trail S*) *K* = *i* + *1* ⟹
T ∼ *cons-trail* (*Propagated L D*)
(*reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None S*)))) ⟹
backtrack S T

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

inductive *decide* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
decide-rule:
conflicting S = None ⟹
undefined-lit (*trail S*) *L* ⟹
atm-of L ∈ *atms-of-mm* (*init-clss S*) ⟹
T ∼ *cons-trail* (*Decided L*) (*incr-lvl S*) ⟹
decide S T

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

inductive *skip* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**

skip-rule:
 trail S = *Propagated* L C' # M $\Longrightarrow$
 conflicting S = *Some* E $\Longrightarrow$
 $-L$ ∉# E $\Longrightarrow$
 $E \neq$ {#} $\Longrightarrow$
 $T \sim$ *tl-trail* S $\Longrightarrow$
 skip S T

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

get-maximum-level (*Propagated* L (C + {#L#}) # M) $D = k \vee k = 0$ (that was in a previous version of the book) is equivalent to *get-maximum-level* (*Propagated* L (C + {#L#}) # M) D = k, when the structural invariants holds.

inductive *resolve* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
resolve-rule: *trail* $S \neq$ [] $\Longrightarrow$
 hd-trail S = *Propagated* L E $\Longrightarrow$
 L ∈# E $\Longrightarrow$
 conflicting S = *Some* D' $\Longrightarrow$
 $-L$ ∈# D' $\Longrightarrow$
 get-maximum-level (*trail* S) ((*remove1-mset* ($-L$) D')) = *backtrack-lvl* S $\Longrightarrow$
 $T \sim$ *update-conflicting* (*Some* (*resolve-cls* L D' E))
 (*tl-trail* S) $\Longrightarrow$
 resolve S T

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

inductive *restart* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
restart: *state* S = (M, N, U, k, *None*) $\Longrightarrow$
 ¬M ⊨*asm clauses* S $\Longrightarrow$
 U' ⊆# U $\Longrightarrow$
 state T = ([], N, U', *0*, *None*) $\Longrightarrow$
 restart S T

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

We add the condition C ∉# *init-clss* S, to maintain consistency even without the strategy.

inductive *forget* :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget-rule:
 conflicting S = *None* $\Longrightarrow$
 C ∈# *learned-clss* S $\Longrightarrow$
 ¬(*trail* S) ⊨*asm clauses* S $\Longrightarrow$
 $C \notin$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\Longrightarrow$
 C ∉# *init-clss* S $\Longrightarrow$
 $T \sim$ *remove-cls* C S $\Longrightarrow$
 forget S T

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

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

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

backtrack: *backtrack* S S' $\Longrightarrow$ $cdcl_W$*-bj* S S'

inductive-cases $cdcl_W$*-bjE*: $cdcl_W$*-bj* S T

inductive $cdcl_W$*-o* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** S :: *'st* **where**
decide: *decide* S S' $\Longrightarrow$ $cdcl_W$*-o* S S' |
bj: $cdcl_W$*-bj* S S' $\Longrightarrow$ $cdcl_W$*-o* S S'

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

lemma *rtranclp-propagate-is-rtranclp-*$cdcl_W$:
  $propagate^{**}$ S S' $\Longrightarrow$ $cdcl_W{}^{**}$ S S'
  ⟨*proof*⟩

lemma $cdcl_W$*-all-rules-induct*[*consumes 1*, *case-names propagate conflict forget restart decide skip resolve backtrack*]:
  fixes S :: *'st*
  assumes
    $cdcl_W$: $cdcl_W$ S S' **and**
    propagate: $\bigwedge T$. *propagate* S T $\Longrightarrow$ P S T **and**
    conflict: $\bigwedge T$. *conflict* S T $\Longrightarrow$ P S T **and**
    forget: $\bigwedge T$. *forget* S T $\Longrightarrow$ P S T **and**
    restart: $\bigwedge T$. *restart* S T $\Longrightarrow$ P S T **and**
    decide: $\bigwedge T$. *decide* S T $\Longrightarrow$ P S T **and**
    skip: $\bigwedge T$. *skip* S T $\Longrightarrow$ P S T **and**
    resolve: $\bigwedge T$. *resolve* S T $\Longrightarrow$ P S T **and**
    backtrack: $\bigwedge T$. *backtrack* S T $\Longrightarrow$ P S T
  shows P S S'
  ⟨*proof*⟩

lemma $cdcl_W$*-all-induct*[*consumes 1*, *case-names propagate conflict forget restart decide skip resolve backtrack*]:
  fixes S :: *'st*
  assumes
    $cdcl_W$: $cdcl_W$ S S' **and**
    propagateH: $\bigwedge C$ L T. *conflicting* S = *None* $\Longrightarrow$
      C $\in\#$ *clauses* S $\Longrightarrow$
      L $\in\#$ C $\Longrightarrow$
      trail S $\models$*as* *CNot* (*remove1-mset* L C) $\Longrightarrow$
      undefined-lit (*trail* S) L $\Longrightarrow$
      T $\sim$ *cons-trail* (*Propagated* L C) S $\Longrightarrow$
      P S T **and**
    conflictH: $\bigwedge D$ T. *conflicting* S = *None* $\Longrightarrow$
      D $\in\#$ *clauses* S $\Longrightarrow$
      trail S $\models$*as* *CNot* D $\Longrightarrow$
      T $\sim$ *update-conflicting* (*Some* D) S $\Longrightarrow$
      P S T **and**
    forgetH: $\bigwedge C$ T. *conflicting* S = *None* $\Longrightarrow$
      C $\in\#$ *learned-clss* S $\Longrightarrow$
      $\neg$(*trail* S) $\models$*asm* *clauses* S $\Longrightarrow$
      C $\notin$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\Longrightarrow$
      C $\notin\#$ *init-clss* S $\Longrightarrow$

$T \sim$ *remove-cls* C $S \Longrightarrow$
$P\ S\ T$ **and**
restartH: $\bigwedge T\ U.\ \neg$*trail* $S \models$*asm clauses* $S \Longrightarrow$
conflicting S = *None* $\Longrightarrow$
state T = ([], *init-clss* S, U, *0*, *None*) $\Longrightarrow$
$U \subseteq\#$ *learned-clss* $S \Longrightarrow$
$P\ S\ T$ **and**
decideH: $\bigwedge L\ T.$ *conflicting* S = *None* $\Longrightarrow$
undefined-lit (*trail* S) $L \Longrightarrow$
atm-of $L \in$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
$T \sim$ *cons-trail* (*Decided* L) (*incr-lvl* S) $\Longrightarrow$
$P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
trail S = *Propagated* L C' # $M \Longrightarrow$
conflicting S = *Some* $E \Longrightarrow$
$-L \notin\#\ E \Longrightarrow E \neq \{\#\} \Longrightarrow$
$T \sim$ *tl-trail* $S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
trail S = *Propagated* L E # $M \Longrightarrow$
$L \in\#\ E \Longrightarrow$
hd-trail S = *Propagated* L $E \Longrightarrow$
conflicting S = *Some* $D \Longrightarrow$
$-L \in\#\ D \Longrightarrow$
get-maximum-level (*trail* S) ((*remove1-mset* $(-L)$ D)) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*resolve-cls* L D E)) (*tl-trail* S) $\Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge L\ D\ K\ i\ M1\ M2\ T.$
conflicting S = *Some* $D \Longrightarrow$
$L \in\#\ D \Longrightarrow$
(*Decided* K # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) $D \Longrightarrow$
get-maximum-level (*trail* S) (*remove1-mset* L D) $\equiv i \Longrightarrow$
get-level (*trail* S) K = i+*1* $\Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* L D)
(*reduce-trail-to M1*
(*add-learned-cls* D
(*update-backtrack-lvl* i
(*update-conflicting None* S)))) $\Longrightarrow$
$P\ S\ T$
shows $P\ S\ S'$
⟨*proof*⟩

lemma $cdcl_W$*-o-induct*[*consumes 1*, *case-names decide skip resolve backtrack*]:
fixes S :: *'st*
assumes $cdcl_W$: $cdcl_W$*-o* S T **and**
decideH: $\bigwedge L\ T.$ *conflicting* S = *None* $\Longrightarrow$ *undefined-lit* (*trail* S) L
$\Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* (*init-clss* S)
$\Longrightarrow T \sim$ *cons-trail* (*Decided* L) (*incr-lvl* S)
$\Longrightarrow P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
trail S = *Propagated* L C' # $M \Longrightarrow$
conflicting S = *Some* $E \Longrightarrow$
$-L \notin\#\ E \Longrightarrow E \neq \{\#\} \Longrightarrow$

$T \sim$ *tl-trail* $S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
trail S = *Propagated* $L\ E\ \#\ M \Longrightarrow$
$L \in\#\ E \Longrightarrow$
hd-trail S = *Propagated* $L\ E \Longrightarrow$
conflicting S = *Some* $D \Longrightarrow$
$-L \in\#\ D \Longrightarrow$
get-maximum-level (*trail* S) ((*remove1-mset* $(-L)\ D$)) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*resolve-cls* $L\ D\ E$)) (*tl-trail* S) $\Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge L\ D\ K\ i\ M1\ M2\ T.$
conflicting S = *Some* $D \Longrightarrow$
$L \in\#\ D \Longrightarrow$
(*Decided* $K\ \#\ M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) $D \Longrightarrow$
get-maximum-level (*trail* S) (*remove1-mset* $L\ D$) $\equiv i \Longrightarrow$
get-level (*trail* S) $K = i + 1 \Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* $L\ D$)
(*reduce-trail-to* $M1$
(*add-learned-cls* D
(*update-backtrack-lvl* i
(*update-conflicting None* S)))) $\Longrightarrow$
$P\ S\ T$
shows $P\ S\ T$
⟨*proof*⟩

thm *cdcl*$_W$*-o.induct*
lemma *cdcl*$_W$*-o-all-rules-induct*[*consumes 1*, *case-names decide backtrack skip resolve*]:
fixes $S\ T$ :: *'st*
assumes
cdcl$_W$*-o* $S\ T$ **and**
$\bigwedge T.$ *decide* $S\ T \Longrightarrow P\ S\ T$ **and**
$\bigwedge T.$ *backtrack* $S\ T \Longrightarrow P\ S\ T$ **and**
$\bigwedge T.$ *skip* $S\ T \Longrightarrow P\ S\ T$ **and**
$\bigwedge T.$ *resolve* $S\ T \Longrightarrow P\ S\ T$
shows $P\ S\ T$
⟨*proof*⟩

lemma *cdcl*$_W$*-o-rule-cases*[*consumes 1*, *case-names decide backtrack skip resolve*]:
fixes $S\ T$ :: *'st*
assumes
cdcl$_W$*-o* $S\ T$ **and**
decide $S\ T \Longrightarrow P$ **and**
backtrack $S\ T \Longrightarrow P$ **and**
skip $S\ T \Longrightarrow P$ **and**
resolve $S\ T \Longrightarrow P$
shows P
⟨*proof*⟩

6.1.3 Structural Invariants

Properties of the trail

We here establish that:

- the consistency of the trail;
- the fact that there is no duplicate in the trail.

lemma *backtrack-lit-skiped*:
assumes
L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
M1: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
no-dup: *no-dup* (*trail S*) **and**
bt-l: *backtrack-lvl S* = *length* (*filter is-decided* (*trail S*)) **and**
lev-K: *get-level* (*trail S*) *K* = *i* + *1*
shows *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
⟨*proof*⟩

lemma $cdcl_W$*-distinctinv-1*:
assumes
$cdcl_W$ *S S'* **and**
no-dup (*trail S*) **and**
bt-lev: *backtrack-lvl S* = *count-decided* (*trail S*)
shows *no-dup* (*trail S'*)
⟨*proof*⟩

Item 1 page 81 of Weidenbach's book

lemma $cdcl_W$*-consistent-inv-2*:
assumes
$cdcl_W$ *S S'* **and**
no-dup (*trail S*) **and**
backtrack-lvl S = *count-decided* (*trail S*)
shows *consistent-interp* (*lits-of-l* (*trail S'*))
⟨*proof*⟩

lemma $cdcl_W$*-o-bt*:
assumes
$cdcl_W$*-o S S'* **and**
backtrack-lvl S = *count-decided* (*trail S*) **and**
n-d[*simp*]: *no-dup* (*trail S*)
shows *backtrack-lvl S'* = *count-decided* (*trail S'*)
⟨*proof*⟩

lemma $cdcl_W$*-rf-bt*:
assumes
$cdcl_W$*-rf S S'* **and**
backtrack-lvl S = *count-decided* (*trail S*)
shows *backtrack-lvl S'* = *count-decided* (*trail S'*)
⟨*proof*⟩

Item 7 page 81 of Weidenbach's book

lemma $cdcl_W$*-bt*:
assumes

$cdcl_W$ S S' **and**
$backtrack\text{-}lvl\ S = count\text{-}decided\ (trail\ S)$ **and**
$no\text{-}dup\ (trail\ S)$
shows $backtrack\text{-}lvl\ S' = count\text{-}decided\ (trail\ S')$
⟨*proof*⟩

We write $1 + count\text{-}decided\ (trail\ S)$ instead of $backtrack\text{-}lvl\ S$ to avoid non termination of rewriting.

definition $cdcl_W\text{-}M\text{-}level\text{-}inv :: 'st \Rightarrow bool$ **where**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ S \longleftrightarrow$
$consistent\text{-}interp\ (lits\text{-}of\text{-}l\ (trail\ S))$
$\wedge\ no\text{-}dup\ (trail\ S)$
$\wedge\ backtrack\text{-}lvl\ S = count\text{-}decided\ (trail\ S)$

lemma $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp$:
assumes $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows
$consistent\text{-}interp\ (lits\text{-}of\text{-}l\ (trail\ S))$ **and**
$no\text{-}dup\ (trail\ S)$
⟨*proof*⟩

lemma $cdcl_W\text{-}consistent\text{-}inv$:
fixes $S\ S' :: 'st$
assumes
$cdcl_W$ S S' **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$
⟨*proof*⟩

lemma $rtranclp\text{-}cdcl_W\text{-}consistent\text{-}inv$:
assumes
$cdcl_W^{**}$ S S' **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$
⟨*proof*⟩

lemma $tranclp\text{-}cdcl_W\text{-}consistent\text{-}inv$:
assumes
$cdcl_W^{++}$ S S' **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$
⟨*proof*⟩

lemma $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}S0\text{-}cdcl_W[simp]$:
$cdcl_W\text{-}M\text{-}level\text{-}inv\ (init\text{-}state\ N)$
⟨*proof*⟩

lemma $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}get\text{-}level\text{-}le\text{-}backtrack\text{-}lvl$:
assumes inv: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $get\text{-}level\ (trail\ S)\ L \leq backtrack\text{-}lvl\ S$
⟨*proof*⟩

lemma $backtrack\text{-}ex\text{-}decomp$:
assumes
$M\text{-}l$: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and**
$i\text{-}S$: $i < backtrack\text{-}lvl\ S$

shows $\exists K\ M1\ M2.\ (Decided\ K\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S)) \wedge$
$get\text{-}level\ (trail\ S)\ K = Suc\ i$
⟨*proof*⟩

lemma *backtrack-lvl-backtrack-decrease*:
assumes *inv*: $cdcl_W$*-M-level-inv S* **and** *bt*: *backtrack S T*
shows *backtrack-lvl T* < *backtrack-lvl S*
⟨*proof*⟩

Compatibility with *op* ∼

lemma *propagate-state-eq-compatible*:
assumes
propa: *propagate S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *propagate S' T'*
⟨*proof*⟩

lemma *conflict-state-eq-compatible*:
assumes
confl: *conflict S T* **and**
TT': $T \sim T'$ **and**
SS': $S \sim S'$
shows *conflict S' T'*
⟨*proof*⟩

lemma *backtrack-state-eq-compatible*:
assumes
bt: *backtrack S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$ **and**
inv: $cdcl_W$*-M-level-inv S*
shows *backtrack S' T'*
⟨*proof*⟩

lemma *decide-state-eq-compatible*:
assumes
decide S T **and**
$S \sim S'$ **and**
$T \sim T'$
shows *decide S' T'*
⟨*proof*⟩

lemma *skip-state-eq-compatible*:
assumes
skip: *skip S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *skip S' T'*
⟨*proof*⟩

lemma *resolve-state-eq-compatible*:
assumes
res: *resolve S T* **and**
TT': $T \sim T'$ **and**

SS': $S \sim S'$
shows *resolve* S' T'
⟨*proof*⟩

lemma *forget-state-eq-compatible*:
assumes
forget: *forget* S T **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *forget* S' T'
⟨*proof*⟩

lemma $cdcl_W$*-state-eq-compatible*:
assumes
$cdcl_W$ S T **and** $\neg$*restart* S T **and**
$S \sim S'$
$T \sim T'$ **and**
$cdcl_W$*-M-level-inv* S
shows $cdcl_W$ S' T'
⟨*proof*⟩

lemma $cdcl_W$*-bj-state-eq-compatible*:
assumes
$cdcl_W$*-bj* S T **and** $cdcl_W$*-M-level-inv* S
$T \sim T'$
shows $cdcl_W$*-bj* S T'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-bj-state-eq-compatible*:
assumes
$cdcl_W$*-bj*$^{++}$ S T **and** *inv*: $cdcl_W$*-M-level-inv* S **and**
$S \sim S'$ **and**
$T \sim T'$
shows $cdcl_W$*-bj*$^{++}$ S' T'
⟨*proof*⟩

Conservation of some Properties

lemma $cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o* S S' **and**
inv: $cdcl_W$*-M-level-inv* S
shows *init-clss* S = *init-clss* S'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o*$^{++}$ S S' **and**
inv: $cdcl_W$*-M-level-inv* S
shows *init-clss* S = *init-clss* S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o*** S S' **and**
inv: $cdcl_W$*-M-level-inv* S

shows $init\text{-}clss\ S = init\text{-}clss\ S'$
⟨*proof*⟩

lemma $cdcl_W\text{-}init\text{-}clss$:
assumes
$cdcl_W\ S\ T$ **and**
inv: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $init\text{-}clss\ S = init\text{-}clss\ T$
⟨*proof*⟩

lemma $rtranclp\text{-}cdcl_W\text{-}init\text{-}clss$:
$cdcl_W^{**}\ S\ T \Longrightarrow cdcl_W\text{-}M\text{-}level\text{-}inv\ S \Longrightarrow init\text{-}clss\ S = init\text{-}clss\ T$
⟨*proof*⟩

lemma $tranclp\text{-}cdcl_W\text{-}init\text{-}clss$:
$cdcl_W^{++}\ S\ T \Longrightarrow cdcl_W\text{-}M\text{-}level\text{-}inv\ S \Longrightarrow init\text{-}clss\ S = init\text{-}clss\ T$
⟨*proof*⟩

Learned Clause

This invariant shows that:

- the learned clauses are entailed by the initial set of clauses.
- the conflicting clause is entailed by the initial set of clauses.
- the marks are entailed by the clauses.

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

of Weidenbach's book for the inital state and some additional structural properties about the trail.

lemma $cdcl_W\text{-}learned\text{-}clause\text{-}S0\text{-}cdcl_W[simp]$:
$cdcl_W\text{-}learned\text{-}clause\ (init\text{-}state\ N)$
⟨*proof*⟩

Item 4 page 81 of Weidenbach's book

lemma $cdcl_W\text{-}learned\text{-}clss$:
assumes
$cdcl_W\ S\ S'$ **and**
learned: $cdcl_W\text{-}learned\text{-}clause\ S$ **and**
lev-inv: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $cdcl_W\text{-}learned\text{-}clause\ S'$
⟨*proof*⟩

lemma $rtranclp\text{-}cdcl_W\text{-}learned\text{-}clss$:
assumes
$cdcl_W^{**}\ S\ S'$ **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
$cdcl_W\text{-}learned\text{-}clause\ S$
shows $cdcl_W\text{-}learned\text{-}clause\ S'$
⟨*proof*⟩

No alien atom in the state

This invariant means that all the literals are in the set of clauses. These properties are implicit in Weidenbach's book.

definition *no-strange-atm* $S' \longleftrightarrow$ (
  ($\forall$ T. *conflicting* S' = *Some* $T \longrightarrow$ *atms-of* $T \subseteq$ *atms-of-mm* (*init-clss* S'))
  $\wedge$ ($\forall$ L *mark*. *Propagated* L *mark* $\in$ *set* (*trail* S')
    $\longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss* S'))
  $\wedge$ *atms-of-mm* (*learned-clss* S') $\subseteq$ *atms-of-mm* (*init-clss* S')
  $\wedge$ *atm-of* ‘ (*lits-of-l* (*trail* S')) $\subseteq$ *atms-of-mm* (*init-clss* S'))

lemma *no-strange-atm-decomp*:
  assumes *no-strange-atm* S
  shows *conflicting* S = *Some* $T \Longrightarrow$ *atms-of* $T \subseteq$ *atms-of-mm* (*init-clss* S)
  and ($\forall$ L *mark*. *Propagated* L *mark* $\in$ *set* (*trail* S)
    $\longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss* S))
  and *atms-of-mm* (*learned-clss* S) $\subseteq$ *atms-of-mm* (*init-clss* S)
  and *atm-of* ‘ (*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-mm* (*init-clss* S)
  ⟨*proof*⟩

lemma *no-strange-atm-S0* [*simp*]: *no-strange-atm* (*init-state* N)
  ⟨*proof*⟩

lemma *in-atms-of-implies-atm-of-on-atms-of-ms*:
  C + {#L#} $\in$# $A \Longrightarrow x \in$ *atms-of* $C \Longrightarrow x \in$ *atms-of-mm* A
  ⟨*proof*⟩

lemma *propagate-no-strange-atm-inv*:
  assumes
    propagate S T **and**
    alien: *no-strange-atm* S
  shows *no-strange-atm* T
  ⟨*proof*⟩

lemma *in-atms-of-remove1-mset-in-atms-of*:
  $x \in$ *atms-of* (*remove1-mset* L C) $\Longrightarrow x \in$ *atms-of* C
  ⟨*proof*⟩

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*:
  atms-of-mm (*learned-clss* S) $\subseteq$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
   $x \in$ *atms-of-mm* (*learned-clss* T) $\Longrightarrow$
   learned-clss T $\subseteq$# *learned-clss* S $\Longrightarrow$
   $x \in$ *atms-of-mm* (*init-clss* S)
  ⟨*proof*⟩

lemma $cdcl_W$*-no-strange-atm-explicit*:
  assumes
    $cdcl_W$ S S' **and**
    lev: $cdcl_W$*-M-level-inv* S **and**
    conf: $\forall$ T. *conflicting* S = *Some* $T \longrightarrow$ *atms-of* $T \subseteq$ *atms-of-mm* (*init-clss* S) **and**
    decided: $\forall$ L *mark*. *Propagated* L *mark* $\in$ *set* (*trail* S)
      $\longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss* S) **and**
    learned: *atms-of-mm* (*learned-clss* S) $\subseteq$ *atms-of-mm* (*init-clss* S) **and**
    trail: *atm-of* ‘ (*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-mm* (*init-clss* S)
  shows

$(\forall T.\ \textit{conflicting } S' = \textit{Some } T \longrightarrow \textit{atms-of } T \subseteq \textit{atms-of-mm } (\textit{init-clss } S')) \land$
$(\forall L\ \textit{mark. Propagated } L\ \textit{mark} \in \textit{set } (\textit{trail } S')$
$\longrightarrow \textit{atms-of mark} \subseteq \textit{atms-of-mm } (\textit{init-clss } S')) \land$
$\textit{atms-of-mm } (\textit{learned-clss } S') \subseteq \textit{atms-of-mm } (\textit{init-clss } S') \land$
$\textit{atm-of } \text{`}\ (\textit{lits-of-l } (\textit{trail } S')) \subseteq \textit{atms-of-mm } (\textit{init-clss } S')$
(**is** $?C\ S' \land\ ?M\ S' \land\ ?U\ S' \land\ ?V\ S'$)
⟨*proof*⟩

lemma $\textit{cdcl}_W$*-no-strange-atm-inv*:
assumes $\textit{cdcl}_W\ S\ S'$ **and** *no-strange-atm S* **and** $\textit{cdcl}_W$*-M-level-inv S*
shows *no-strange-atm S′*
⟨*proof*⟩

lemma *rtranclp-*$\textit{cdcl}_W$*-no-strange-atm-inv*:
assumes $\textit{cdcl}_W{}^{**}\ S\ S'$ **and** *no-strange-atm S* **and** $\textit{cdcl}_W$*-M-level-inv S*
shows *no-strange-atm S′*
⟨*proof*⟩

No Duplicates all Around

This invariant shows that there is no duplicate (no literal appearing twice in the formula). The last part could be proven using the previous invariant also. Remark that we will show later that there cannot be duplicate *clause*.

definition *distinct-*$\textit{cdcl}_W$*-state* $(S :: 'st)$
$\longleftrightarrow ((\forall T.\ \textit{conflicting } S = \textit{Some } T \longrightarrow \textit{distinct-mset } T)$
$\land\ \textit{distinct-mset-mset } (\textit{learned-clss } S)$
$\land\ \textit{distinct-mset-mset } (\textit{init-clss } S)$
$\land\ (\forall L\ \textit{mark}.\ (\textit{Propagated } L\ \textit{mark} \in \textit{set } (\textit{trail } S) \longrightarrow \textit{distinct-mset mark})))$

lemma *distinct-*$\textit{cdcl}_W$*-state-decomp*:
assumes *distinct-*$\textit{cdcl}_W$*-state* $(S :: 'st)$
shows
$\forall T.\ \textit{conflicting } S = \textit{Some } T \longrightarrow \textit{distinct-mset } T$ **and**
$\textit{distinct-mset-mset } (\textit{learned-clss } S)$ **and**
$\textit{distinct-mset-mset } (\textit{init-clss } S)$ **and**
$\forall L\ \textit{mark}.\ (\textit{Propagated } L\ \textit{mark} \in \textit{set } (\textit{trail } S) \longrightarrow \textit{distinct-mset mark})$
⟨*proof*⟩

lemma *distinct-*$\textit{cdcl}_W$*-state-decomp-2*:
assumes *distinct-*$\textit{cdcl}_W$*-state* $(S :: 'st)$ **and** $\textit{conflicting } S = \textit{Some } T$
shows *distinct-mset T*
⟨*proof*⟩

lemma *distinct-*$\textit{cdcl}_W$*-state-S0-*$\textit{cdcl}_W$[*simp*]:
$\textit{distinct-mset-mset } N \Longrightarrow$ *distinct-*$\textit{cdcl}_W$*-state* $(\textit{init-state } N)$
⟨*proof*⟩

lemma *distinct-*$\textit{cdcl}_W$*-state-inv*:
assumes
$\textit{cdcl}_W\ S\ S'$ **and**
lev-inv: $\textit{cdcl}_W$*-M-level-inv S* **and**
distinct-$\textit{cdcl}_W$*-state S*
shows *distinct-*$\textit{cdcl}_W$*-state S′*
⟨*proof*⟩

lemma *rtanclp-distinct-cdcl$_W$-state-inv*:
assumes
cdcl$_W$** S S' **and**
cdcl$_W$-M-level-inv S **and**
distinct-cdcl$_W$-state S
shows *distinct-cdcl$_W$-state* S'
⟨*proof*⟩

Conflicts and Annotations

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

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

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

lemma *backtrack-atms-of-D-in-M1*:
fixes *M1* :: (*'v*, *'v clause*) *ann-lits*
assumes
inv: *cdcl$_W$-M-level-inv* S **and**
i: *get-maximum-level* (*trail* S) ((*remove1-mset* L D)) ≡ i **and**
decomp: (*Decided* K # *M1*, *M2*)
∈ *set* (*get-all-ann-decomposition* (*trail* S)) **and**
S-lvl: *backtrack-lvl* S = *get-maximum-level* (*trail* S) D **and**
S-confl: *conflicting* S = *Some* D **and**
lev-K: *get-level* (*trail* S) K = *Suc* i **and**
T: T ∼ *cons-trail* (*Propagated* L D)
(*reduce-trail-to* *M1*
(*add-learned-cls* D
(*update-backtrack-lvl* i
(*update-conflicting* *None* S)))) **and**
confl: ∀ T. *conflicting* S = *Some* T ⟶ *trail* S ⊨*as* *CNot* T
shows *atms-of* ((*remove1-mset* L D)) ⊆ *atm-of* ‘ *lits-of-l* (*tl* (*trail* T))
⟨*proof*⟩

lemma *distinct-atms-of-incl-not-in-other*:
assumes
a1: *no-dup* (M @ M') **and**
a2: *atms-of* D ⊆ *atm-of* ‘ *lits-of-l* M' **and**
a3: x ∈ *atms-of* D
shows x ∉ *atm-of* ‘ *lits-of-l* M
⟨*proof*⟩

Item 5 page 81 of Weidenbach's book

lemma *cdcl$_W$-propagate-is-conclusion*:
assumes
cdcl$_W$ S S' **and**
inv: *cdcl$_W$-M-level-inv* S **and**
decomp: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
learned: *cdcl$_W$-learned-clause* S **and**

confl: $\forall T$. *conflicting* S = *Some* $T \longrightarrow$ *trail* $S \models as$ *CNot* T **and**
alien: *no-strange-atm* S
shows *all-decomposition-implies-m* (*init-clss* S') (*get-all-ann-decomposition* (*trail* S'))
⟨*proof*⟩

lemma *cdcl*$_W$*-propagate-is-false*:
assumes
cdcl$_W$ S S' **and**
lev: *cdcl*$_W$*-M-level-inv* S **and**
learned: *cdcl*$_W$*-learned-clause* S **and**
decomp: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
confl: $\forall T$. *conflicting* S = *Some* $T \longrightarrow$ *trail* $S \models as$ *CNot* T **and**
alien: *no-strange-atm* S **and**
mark-confl: *every-mark-is-a-conflict* S
shows *every-mark-is-a-conflict* S'
⟨*proof*⟩

lemma *cdcl*$_W$*-conflicting-is-false*:
assumes
cdcl$_W$ S S' **and**
M-lev: *cdcl*$_W$*-M-level-inv* S **and**
confl-inv: $\forall T$. *conflicting* S = *Some* $T \longrightarrow$ *trail* $S \models as$ *CNot* T **and**
decided-confl: $\forall L$ *mark a b*. *a* @ *Propagated L mark* # *b* = (*trail* S)
$\longrightarrow$ (*b* $\models as$ *CNot* (*mark* − {#*L*#}) ∧ *L* ∈# *mark*) **and**
dist: *distinct-cdcl*$_W$*-state* S
shows $\forall T$. *conflicting* S' = *Some* $T \longrightarrow$ *trail* $S' \models as$ *CNot* T
⟨*proof*⟩

lemma *cdcl*$_W$*-conflicting-decomp*:
assumes *cdcl*$_W$*-conflicting* S
shows $\forall T$. *conflicting* S = *Some* $T \longrightarrow$ *trail* $S \models as$ *CNot* T
and $\forall L$ *mark a b*. *a* @ *Propagated L mark* # *b* = (*trail* S)
$\longrightarrow$ (*b* $\models as$ *CNot* (*mark* − {#*L*#}) ∧ *L* ∈# *mark*)
⟨*proof*⟩

lemma *cdcl*$_W$*-conflicting-decomp2*:
assumes *cdcl*$_W$*-conflicting* S **and** *conflicting* S = *Some* T
shows *trail* $S \models as$ *CNot* T
⟨*proof*⟩

lemma *cdcl*$_W$*-conflicting-S0-cdcl*$_W$[*simp*]:
cdcl$_W$*-conflicting* (*init-state* N)
⟨*proof*⟩

Putting all the invariants together

lemma *cdcl*$_W$*-all-inv*:
assumes
cdcl$_W$: *cdcl*$_W$ S S' **and**
1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
2: *cdcl*$_W$*-learned-clause* S **and**
4: *cdcl*$_W$*-M-level-inv* S **and**
5: *no-strange-atm* S **and**
7: *distinct-cdcl*$_W$*-state* S **and**
8: *cdcl*$_W$*-conflicting* S
shows

all-decomposition-implies-m (*init-clss* S') (*get-all-ann-decomposition* (*trail* S')) **and**
$cdcl_W$*-learned-clause* S' **and**
$cdcl_W$*-M-level-inv* S' **and**
no-strange-atm S' **and**
distinct-$cdcl_W$*-state* S' **and**
$cdcl_W$*-conflicting* S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-all-inv*:
assumes
$cdcl_W$: *rtranclp* $cdcl_W$ S S' **and**
1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
2: $cdcl_W$*-learned-clause* S **and**
4: $cdcl_W$*-M-level-inv* S **and**
5: *no-strange-atm* S **and**
7: *distinct-*$cdcl_W$*-state* S **and**
8: $cdcl_W$*-conflicting* S
shows
all-decomposition-implies-m (*init-clss* S') (*get-all-ann-decomposition* (*trail* S')) **and**
$cdcl_W$*-learned-clause* S' **and**
$cdcl_W$*-M-level-inv* S' **and**
no-strange-atm S' **and**
distinct-$cdcl_W$*-state* S' **and**
$cdcl_W$*-conflicting* S'
⟨*proof*⟩

lemma *all-invariant-S0-*$cdcl_W$:
assumes *distinct-mset-mset* N
shows
all-decomposition-implies-m (*init-clss* (*init-state* N))
(*get-all-ann-decomposition* (*trail* (*init-state* N))) **and**
$cdcl_W$*-learned-clause* (*init-state* N) **and**
$\forall$ T. *conflicting* (*init-state* N) = *Some* T $\longrightarrow$ (*trail* (*init-state* N))$\models$*as* *CNot* T **and**
no-strange-atm (*init-state* N) **and**
consistent-interp (*lits-of-l* (*trail* (*init-state* N))) **and**
$\forall$ *L mark a b*. *a* @ *Propagated L mark* # *b* = *trail* (*init-state* N) $\longrightarrow$
(*b* $\models$*as CNot* (*mark* − {#*L*#}) $\wedge$ *L* $\in$# *mark*) **and**
distinct-$cdcl_W$*-state* (*init-state* N)
⟨*proof*⟩

Item 6 page 81 of Weidenbach's book

lemma $cdcl_W$*-only-propagated-vars-unsat*:
assumes
decided: $\forall$ $x \in$ *set* M. $\neg$ *is-decided* x **and**
DN: D $\in$# *clauses* S **and**
D: M $\models$*as CNot* D **and**
inv: *all-decomposition-implies-m* N (*get-all-ann-decomposition* M) **and**
state: *state* S = (M, N, U, k, C) **and**
learned-cl: $cdcl_W$*-learned-clause* S **and**
atm-incl: *no-strange-atm* S
shows *unsatisfiable* (*set-mset* N)
⟨*proof*⟩

Item 5 page 81 of Weidenbach's book

We have actually a much stronger theorem, namely *all-decomposition-implies-propagated-lits-are-implied*, that show that the only choices we made are decided in the formula

lemma
assumes *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
and $\forall\, m \in$ *set* M. $\neg$*is-decided* m
shows *set-mset* N $\models$*ps unmark-l* M
⟨*proof*⟩

Item 7 page 81 of Weidenbach's book (part 1)

lemma *conflict-with-false-implies-unsat*:
assumes
$cdcl_W$: $cdcl_W$ S S' **and**
lev: $cdcl_W$*-M-level-inv* S **and**
[*simp*]: *conflicting* S' = *Some* {#} **and**
learned: $cdcl_W$*-learned-clause* S
shows *unsatisfiable* (*set-mset* (*init-clss* S))
⟨*proof*⟩

Item 7 page 81 of Weidenbach's book (part 2)

lemma *conflict-with-false-implies-terminated*:
assumes $cdcl_W$ S S'
and *conflicting* S = *Some* {#}
shows *False*
⟨*proof*⟩

No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

lemma *learned-clss-are-not-tautologies*:
assumes
$cdcl_W$ S S' **and**
lev: $cdcl_W$*-M-level-inv* S **and**
conflicting: $cdcl_W$*-conflicting* S **and**
no-tauto: $\forall\, s$ ∈# *learned-clss* S. $\neg$*tautology* s
shows $\forall\, s$ ∈# *learned-clss* S'. $\neg$*tautology* s
⟨*proof*⟩

definition *final-*$cdcl_W$*-state* (S :: *'st*)
$\longleftrightarrow$ (*trail* S $\models$*asm init-clss* S
$\vee$ (($\forall\, L \in$ *set* (*trail* S). $\neg$*is-decided* L) $\wedge$
($\exists\, C$ ∈# *init-clss* S. *trail* S $\models$*as CNot* C)))

definition *termination-*$cdcl_W$*-state* (S :: *'st*)
$\longleftrightarrow$ (*trail* S $\models$*asm init-clss* S
$\vee$ (($\forall\, L \in$ *atms-of-mm* (*init-clss* S). $L \in$ *atm-of* ‘ *lits-of-l* (*trail* S))
$\wedge$ ($\exists\, C$ ∈# *init-clss* S. *trail* S $\models$*as CNot* C)))

6.1.4 CDCL Strong Completeness

lemma $cdcl_W$*-can-do-step*:
assumes
consistent-interp (*set* M) **and**
distinct M **and**
atm-of ‘ (*set* M) $\subseteq$ *atms-of-mm* N

shows $\exists S.$ *rtranclp* $cdcl_W$ (*init-state* N) S
$\wedge$ *state* S = (*map* ($\lambda L.$ *Decided* L) M, N, {#}, *length* M, *None*)
⟨*proof*⟩

theorem 2.9.11 page 84 of Weidenbach's book

lemma $cdcl_W$*-strong-completeness*:
assumes
MN: *set* M $\models$*sm* N **and**
cons: *consistent-interp* (*set* M) **and**
dist: *distinct* M **and**
atm: *atm-of* ‘ (*set* M) $\subseteq$ *atms-of-mm* N
obtains S **where**
state S = (*map* ($\lambda L.$ *Decided* L) M, N, {#}, *length* M, *None*) **and**
rtranclp $cdcl_W$ (*init-state* N) S **and**
final-$cdcl_W$*-state* S
⟨*proof*⟩

6.1.5 Higher level strategy

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

Definition

lemma *tranclp-conflict*:
tranclp conflict S S' $\Longrightarrow$ *conflict* S S'
⟨*proof*⟩

lemma *tranclp-conflict-iff* [*iff*]:
full1 conflict S S' $\longleftrightarrow$ *conflict* S S'
⟨*proof*⟩

inductive $cdcl_W$*-cp* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **where**
conflict'[*intro*]: *conflict* S S' $\Longrightarrow$ $cdcl_W$*-cp* S S' |
propagate': *propagate* S S' $\Longrightarrow$ $cdcl_W$*-cp* S S'

lemma *rtranclp-*$cdcl_W$*-cp-rtranclp-*$cdcl_W$:
$cdcl_W$*-cp*** S T $\Longrightarrow$ $cdcl_W{}^{**}$ S T
⟨*proof*⟩

lemma $cdcl_W$*-cp-state-eq-compatible*:
assumes
$cdcl_W$*-cp* S T **and**
$S \sim S'$ **and**
$T \sim T'$
shows $cdcl_W$*-cp* S' T'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-cp-state-eq-compatible*:
assumes
$cdcl_W$*-cp*$^{++}$ S T **and**
$S \sim S'$ **and**
$T \sim T'$
shows $cdcl_W$*-cp*$^{++}$ S' T'
⟨*proof*⟩

lemma *option-full-cdcl$_W$-cp*:
conflicting $S \neq$ *None* $\Longrightarrow$ *full cdcl$_W$-cp S S*
⟨*proof*⟩

lemma *skip-unique*:
skip S $T \Longrightarrow$ *skip* S $T' \Longrightarrow T \sim T'$
⟨*proof*⟩

lemma *resolve-unique*:
resolve S $T \Longrightarrow$ *resolve* S $T' \Longrightarrow T \sim T'$
⟨*proof*⟩

lemma *cdcl$_W$-cp-no-more-clauses*:
assumes *cdcl$_W$-cp* S S'
shows *clauses* S = *clauses* S'
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-no-more-clauses*:
assumes *cdcl$_W$-cp^{++}* S S'
shows *clauses* S = *clauses* S'
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-no-more-clauses*:
assumes *cdcl$_W$-cp*** S S'
shows *clauses* S = *clauses* S'
⟨*proof*⟩

lemma *no-conflict-after-conflict*:
conflict S $T \Longrightarrow \neg$*conflict* T U
⟨*proof*⟩

lemma *no-propagate-after-conflict*:
conflict S $T \Longrightarrow \neg$*propagate* T U
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-propagate-with-conflict-or-not*:
assumes *cdcl$_W$-cp^{++}* S U
shows (*propagate^{++}* S U $\wedge$ *conflicting* U = *None*)
$\vee$ ($\exists$ T D. *propagate*** S T $\wedge$ *conflict* T U $\wedge$ *conflicting* U = *Some* D)
⟨*proof*⟩

lemma *cdcl$_W$-cp-conflicting-not-empty*[*simp*]: *conflicting* S = *Some* $D \Longrightarrow \neg$*cdcl$_W$-cp* S S'
⟨*proof*⟩

lemma *no-step-cdcl$_W$-cp-no-conflict-no-propagate*:
assumes *no-step cdcl$_W$-cp* S
shows *no-step conflict* S **and** *no-step propagate* S
⟨*proof*⟩

CDCL with the reasonable strategy: we fully propagate the conflict and propagate, then we apply any other possible rule *cdcl$_W$-o* S S' and re-apply conflict and propagate *cdcl$_W$-cp$^{\downarrow}$* S' S''

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

Invariants

These are the same invariants as before, but lifted

lemma *cdcl*$_W$*-cp-learned-clause-inv*:
assumes *cdcl*$_W$*-cp* S S'
shows *learned-clss* S = *learned-clss* S'
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-learned-clause-inv*:
assumes *cdcl*$_W$*-cp*** S S'
shows *learned-clss* S = *learned-clss* S'
⟨*proof*⟩

lemma *tranclp-cdcl*$_W$*-cp-learned-clause-inv*:
assumes *cdcl*$_W$*-cp*$^{++}$ S S'
shows *learned-clss* S = *learned-clss* S'
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-backtrack-lvl*:
assumes *cdcl*$_W$*-cp* S S'
shows *backtrack-lvl* S = *backtrack-lvl* S'
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-backtrack-lvl*:
assumes *cdcl*$_W$*-cp*** S S'
shows *backtrack-lvl* S = *backtrack-lvl* S'
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-consistent-inv*:
assumes *cdcl*$_W$*-cp* S S' **and** *cdcl*$_W$*-M-level-inv* S
shows *cdcl*$_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *full1-cdcl*$_W$*-cp-consistent-inv*:
assumes *full1 cdcl*$_W$*-cp* S S' **and** *cdcl*$_W$*-M-level-inv* S
shows *cdcl*$_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-consistent-inv*:
assumes *rtranclp cdcl*$_W$*-cp* S S' **and** *cdcl*$_W$*-M-level-inv* S
shows *cdcl*$_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-consistent-inv*:
assumes *cdcl*$_W$*-stgy* S S' **and** *cdcl*$_W$*-M-level-inv* S
shows *cdcl*$_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-consistent-inv*:
assumes *cdcl*$_W$*-stgy*** S S' **and** *cdcl*$_W$*-M-level-inv* S
shows *cdcl*$_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-no-more-init-clss*:
assumes *cdcl*$_W$*-cp* S S'

shows *init-clss* S = *init-clss* S'
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-no-more-init-clss*:
assumes *cdcl$_W$-cp*$^{++}$ S S'
shows *init-clss* S = *init-clss* S'
⟨*proof*⟩

lemma *cdcl$_W$-stgy-no-more-init-clss*:
assumes *cdcl$_W$-stgy* S S' **and** *cdcl$_W$-M-level-inv* S
shows *init-clss* S = *init-clss* S'
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-stgy-no-more-init-clss*:
assumes *cdcl$_W$-stgy*** S S' **and** *cdcl$_W$-M-level-inv* S
shows *init-clss* S = *init-clss* S'
⟨*proof*⟩

lemma *cdcl$_W$-cp-dropWhile-trail'*:
assumes *cdcl$_W$-cp* S S'
obtains M **where** *trail* S' = M @ *trail* S **and** ($\forall$ l $\in$ *set* M. $\neg$*is-decided* l)
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail'*:
assumes *cdcl$_W$-cp*** S S'
obtains M :: (*'v*, *'v clause*) *ann-lits* **where**
trail S' = M @ *trail* S **and** $\forall$ l $\in$ *set* M. $\neg$*is-decided* l
⟨*proof*⟩

lemma *cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp* S S'
shows $\exists$ M. *trail* S' = M @ *trail* S $\wedge$ ($\forall$ l $\in$ *set* M. $\neg$*is-decided* l)
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp*** S S'
shows $\exists$ M. *trail* S' = M @ *trail* S $\wedge$ ($\forall$ l $\in$ *set* M. $\neg$*is-decided* l)
⟨*proof*⟩

This theorem can be seen a a termination theorem for *cdcl$_W$-cp*.

lemma *length-model-le-vars*:
assumes
no-strange-atm S **and**
no-d: *no-dup* (*trail* S) **and**
finite (*atms-of-mm* (*init-clss* S))
shows *length* (*trail* S) $\leq$ *card* (*atms-of-mm* (*init-clss* S))
⟨*proof*⟩

lemma *cdcl$_W$-cp-decreasing-measure*:
assumes
cdcl$_W$: *cdcl$_W$-cp* S T **and**
M-lev: *cdcl$_W$-M-level-inv* S **and**
alien: *no-strange-atm* S
shows (λS. *card* (*atms-of-mm* (*init-clss* S)) $-$ *length* (*trail* S)
$+$ (*if conflicting* S = *None then 1 else 0*)) S
$>$ (λS. *card* (*atms-of-mm* (*init-clss* S)) $-$ *length* (*trail* S)

$+$ (*if conflicting S* $=$ *None then 1 else 0*)) *T*
⟨*proof*⟩

lemma $cdcl_W$*-cp-wf*: *wf* {(*b*, *a*). ($cdcl_W$*-M-level-inv a* $\wedge$ *no-strange-atm a*) $\wedge$ $cdcl_W$*-cp a b*}
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-all-struct-inv-*$cdcl_W$*-cp-iff-rtranclp-*$cdcl_W$*-cp*:
assumes
lev: $cdcl_W$*-M-level-inv S* **and**
alien: *no-strange-atm S*
shows (λa *b*. ($cdcl_W$*-M-level-inv a* $\wedge$ *no-strange-atm a*) $\wedge$ $cdcl_W$*-cp a b*)** *S T*
$\longleftrightarrow$ $cdcl_W$*-cp*** *S T*
(**is** *?I S T* $\longleftrightarrow$ *?C S T*)
⟨*proof*⟩

lemma $cdcl_W$*-cp-normalized-element*:
assumes
lev: $cdcl_W$*-M-level-inv S* **and**
no-strange-atm S
obtains *T* **where** *full* $cdcl_W$*-cp S T*
⟨*proof*⟩

lemma *always-exists-full-*$cdcl_W$*-cp-step*:
assumes *no-strange-atm S*
shows $\exists$ *S''*. *full* $cdcl_W$*-cp S S''*
⟨*proof*⟩

Literal of highest level in conflicting clauses

One important property of the $cdcl_W$ with strategy is that, whenever a conflict takes place, there is at least a literal of level k involved (except if we have derived the false clause). The reason is that we apply conflicts before a decision is taken.

abbreviation *no-clause-is-false* :: *'st* $\Rightarrow$ *bool* **where**
no-clause-is-false $\equiv$
λS. (*conflicting S* $=$ *None* $\longrightarrow$ ($\forall$ *D* $\in\#$ *clauses S*. $\neg$*trail S* $\models$*as CNot D*))

abbreviation *conflict-is-false-with-level* :: *'st* $\Rightarrow$ *bool* **where**
conflict-is-false-with-level S $\equiv$ $\forall$ *D*. *conflicting S* $=$ *Some D* $\longrightarrow$ *D* $\neq$ {#}
$\longrightarrow$ ($\exists$ *L* $\in\#$ *D*. *get-level* (*trail S*) *L* $=$ *backtrack-lvl S*)

lemma *not-conflict-not-any-negated-init-clss*:
assumes $\forall$ *S'*. $\neg$*conflict S S'*
shows *no-clause-is-false S*
⟨*proof*⟩

lemma *full-*$cdcl_W$*-cp-not-any-negated-init-clss*:
assumes *full* $cdcl_W$*-cp S S'*
shows *no-clause-is-false S'*
⟨*proof*⟩

lemma *full1-*$cdcl_W$*-cp-not-any-negated-init-clss*:
assumes *full1* $cdcl_W$*-cp S S'*
shows *no-clause-is-false S'*
⟨*proof*⟩

lemma *$cdcl_W$-stgy-not-non-negated-init-clss*:
assumes *$cdcl_W$-stgy S S'*
shows *no-clause-is-false S'*
⟨*proof*⟩

lemma *rtranclp-$cdcl_W$-stgy-not-non-negated-init-clss*:
assumes *$cdcl_W$-stgy** S S'* **and** *no-clause-is-false S*
shows *no-clause-is-false S'*
⟨*proof*⟩

lemma *$cdcl_W$-stgy-conflict-ex-lit-of-max-level*:
assumes
$cdcl_W$-cp S S' **and**
no-clause-is-false S **and**
$cdcl_W$-M-level-inv S
shows *conflict-is-false-with-level S'*
⟨*proof*⟩

lemma *no-chained-conflict*:
assumes *conflict S S'* **and** *conflict S' S''*
shows *False*
⟨*proof*⟩

lemma *rtranclp-$cdcl_W$-cp-propa-or-propa-confl*:
assumes *$cdcl_W$-cp** S U*
shows $propagate^{**}\ S\ U \lor (\exists\, T.\ propagate^{**}\ S\ T \land conflict\ T\ U)$
⟨*proof*⟩

lemma *rtranclp-$cdcl_W$-co-conflict-ex-lit-of-max-level*:
assumes *full*: *full $cdcl_W$-cp S U*
and *cls-f*: *no-clause-is-false S*
and *conflict-is-false-with-level S*
and *lev*: *$cdcl_W$-M-level-inv S*
shows *conflict-is-false-with-level U*
⟨*proof*⟩

Literal of highest level in decided literals

definition *mark-is-false-with-level* :: *'st ⇒ bool* **where**
mark-is-false-with-level S' ≡
$\forall D\ M1\ M2\ L.\ M1\ @\ Propagated\ L\ D\ \#\ M2 = trail\ S' \longrightarrow D - \{\#L\#\} \neq \{\#\}$
$\longrightarrow (\exists\, L.\ L \in\#\ D \land get\text{-}level\ (trail\ S')\ L = count\text{-}decided\ M1)$

definition *no-more-propagation-to-do* :: *'st ⇒ bool* **where**
no-more-propagation-to-do S ≡
$\forall D\ M\ M'\ L.\ D + \{\#L\#\} \in\#\ clauses\ S \longrightarrow trail\ S = M'\ @\ M \longrightarrow M \models as\ CNot\ D$
$\longrightarrow undefined\text{-}lit\ M\ L \longrightarrow count\text{-}decided\ M < backtrack\text{-}lvl\ S$
$\longrightarrow (\exists\, L.\ L \in\#\ D \land get\text{-}level\ (trail\ S)\ L = count\text{-}decided\ M)$

lemma *propagate-no-more-propagation-to-do*:
assumes *propagate*: *propagate S S'*
and *H*: *no-more-propagation-to-do S*
and *lev-inv*: *$cdcl_W$-M-level-inv S*
shows *no-more-propagation-to-do S'*
⟨*proof*⟩

lemma *conflict-no-more-propagation-to-do*:
  assumes
    conflict: *conflict* S S' **and**
    H: *no-more-propagation-to-do* S **and**
    M: $cdcl_W$*-M-level-inv* S
  shows *no-more-propagation-to-do* S'
  ⟨*proof*⟩

lemma $cdcl_W$*-cp-no-more-propagation-to-do*:
  assumes
    conflict: $cdcl_W$*-cp* S S' **and**
    H: *no-more-propagation-to-do* S **and**
    M: $cdcl_W$*-M-level-inv* S
  shows *no-more-propagation-to-do* S'
  ⟨*proof*⟩

lemma $cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step*:
  assumes
    o: $cdcl_W$*-o* S S' **and**
    alien: *no-strange-atm* S **and**
    lev: $cdcl_W$*-M-level-inv* S
  shows $\exists S'$. $cdcl_W$*-stgy* S S'
⟨*proof*⟩

lemma *backtrack-no-decomp*:
  assumes
    S: *conflicting* S = *Some* E **and**
    LE: $L \in\# E$ **and**
    L: *get-level* (*trail* S) L = *backtrack-lvl* S **and**
    D: *get-maximum-level* (*trail* S) (*remove1-mset* L E) < *backtrack-lvl* S **and**
    bt: *backtrack-lvl* S = *get-maximum-level* (*trail* S) E **and**
    M-L: $cdcl_W$*-M-level-inv* S
  shows $\exists S'$. $cdcl_W$*-o* S S'
⟨*proof*⟩

lemma $cdcl_W$*-stgy-final-state-conclusive*:
  assumes
    termi: $\forall S'$. $\neg$$cdcl_W$*-stgy* S S' **and**
    decomp: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
    learned: $cdcl_W$*-learned-clause* S **and**
    level-inv: $cdcl_W$*-M-level-inv* S **and**
    alien: *no-strange-atm* S **and**
    no-dup: *distinct-*$cdcl_W$*-state* S **and**
    confl: $cdcl_W$*-conflicting* S **and**
    confl-k: *conflict-is-false-with-level* S
  shows (*conflicting* S = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss* S)))
      ∨ (*conflicting* S = *None* ∧ *trail* S ⊨*as set-mset* (*init-clss* S))
⟨*proof*⟩

lemma $cdcl_W$*-cp-tranclp-*$cdcl_W$:
  $cdcl_W$*-cp* S S' ⟹ $cdcl_W{}^{++}$ S S'
  ⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-cp-tranclp-*$cdcl_W$:
  $cdcl_W$*-cp*$^{++}$ S S' ⟹ $cdcl_W{}^{++}$ S S'
  ⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-tranclp-cdcl*$_W$:
  cdcl$_W$*-stgy* S $S' \Longrightarrow$ *cdcl*$_W{}^{++}$ S S'
⟨*proof*⟩

lemma *tranclp-cdcl*$_W$*-stgy-tranclp-cdcl*$_W$:
 cdcl$_W$*-stgy*$^{++}$ S $S' \Longrightarrow$ *cdcl*$_W{}^{++}$ S S'
 ⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-rtranclp-cdcl*$_W$:
  cdcl$_W$*-stgy*** S $S' \Longrightarrow$ *cdcl*$_W{}^{**}$ S S'
 ⟨*proof*⟩

lemma *not-empty-get-maximum-level-exists-lit*:
 assumes *n*: $D \neq \{\#\}$
 and *max*: *get-maximum-level* M $D = n$
 shows $\exists\, L \in\# D$. *get-level* M $L = n$
⟨*proof*⟩

lemma *cdcl*$_W$*-o-conflict-is-false-with-level-inv*:
 assumes
  cdcl$_W$*-o* S S' **and**
  lev: *cdcl*$_W$*-M-level-inv* S **and**
  confl-inv: *conflict-is-false-with-level* S **and**
  n-d: *distinct-cdcl*$_W$*-state* S **and**
  conflicting: *cdcl*$_W$*-conflicting* S
 shows *conflict-is-false-with-level* S'
 ⟨*proof*⟩

Strong completeness

lemma *cdcl*$_W$*-cp-propagate-confl*:
 assumes *cdcl*$_W$*-cp* S T
 shows *propagate*** S T $\vee$ ($\exists\, S'$. *propagate*** S S' $\wedge$ *conflict* S' T)
 ⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-propagate-confl*:
 assumes *cdcl*$_W$*-cp*** S T
 shows *propagate*** S T $\vee$ ($\exists\, S'$. *propagate*** S S' $\wedge$ *conflict* S' T)
 ⟨*proof*⟩

lemma *propagate-high-levelE*:
 assumes *propagate* S T
 obtains M' N' U k L C **where**
  state $S = (M', N', U, k, None)$ **and**
  state $T = (Propagated\ L\ (C + \{\#L\#\})\ \#\ M', N', U, k, None)$ **and**
  $C + \{\#L\#\} \in\#$ *local.clauses* S **and**
  $M' \models as$ *CNot* C **and**
  undefined-lit (*trail* S) L
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-propagate-completeness*:
 assumes *MN*: *set* M $\models s$ *set-mset* N **and**
 cons: *consistent-interp* (*set* M) **and**
 tot: *total-over-m* (*set* M) (*set-mset* N) **and**
 lits-of-l (*trail* S) $\subseteq$ *set* M **and**

init-clss $S = N$ **and**
propagate** S S' **and**
learned-clss $S = \{\#\}$
shows *length* (*trail* S) $\leq$ *length* (*trail* S') $\wedge$ *lits-of-l* (*trail* S') $\subseteq$ *set* M
$\langle$*proof*$\rangle$

lemma
assumes *propagate*** S X
shows
rtranclp-propagate-init-clss: *init-clss* X = *init-clss* S **and**
rtranclp-propagate-learned-clss: *learned-clss* X = *learned-clss* S
$\langle$*proof*$\rangle$

lemma *completeness-is-a-full1-propagation*:
fixes S :: *'st* **and** M :: *'v literal list*
assumes *MN*: *set* M $\models$*s* *set-mset* N
and *cons*: *consistent-interp* (*set* M)
and *tot*: *total-over-m* (*set* M) (*set-mset* N)
and *alien*: *no-strange-atm* S
and *learned*: *learned-clss* $S = \{\#\}$
and *clsS*[*simp*]: *init-clss* $S = N$
and *lits*: *lits-of-l* (*trail* S) $\subseteq$ *set* M
shows $\exists S'$. *propagate*** S S' $\wedge$ *full* $cdcl_W$*-cp* S S'
$\langle$*proof*$\rangle$

See also theorem *rtranclp-*$cdcl_W$*-cp-dropWhile-trail*

lemma *rtranclp-propagate-is-trail-append*:
propagate** S T $\Longrightarrow$ $\exists c$. *trail* $T = c$ @ *trail* S
$\langle$*proof*$\rangle$

lemma *rtranclp-propagate-is-update-trail*:
propagate** S T $\Longrightarrow$ $cdcl_W$*-M-level-inv* S $\Longrightarrow$
init-clss S = *init-clss* T $\wedge$ *learned-clss* S = *learned-clss* T $\wedge$ *backtrack-lvl* S = *backtrack-lvl* T
$\wedge$ *conflicting* S = *conflicting* T
$\langle$*proof*$\rangle$

lemma $cdcl_W$*-stgy-strong-completeness-n*:
assumes
MN: *set* M $\models$*s* *set-mset* N **and**
cons: *consistent-interp* (*set* M) **and**
tot: *total-over-m* (*set* M) (*set-mset* N) **and**
atm-incl: *atm-of* ‘ (*set* M) $\subseteq$ *atms-of-mm* N **and**
distM: *distinct* M **and**
length: $n \leq$ *length* M
shows
$\exists M'$ k S. *length* $M' \geq n$ $\wedge$
lits-of-l M' $\subseteq$ *set* M $\wedge$
no-dup M' $\wedge$
state S = (M', N, $\{\#\}$, k, *None*) $\wedge$
$cdcl_W$*-stgy*** (*init-state* N) S
$\langle$*proof*$\rangle$

theorem 2.9.11 page 84 of Weidenbach's book (with strategy)

lemma $cdcl_W$*-stgy-strong-completeness*:
assumes
MN: *set* M $\models$*s* *set-mset* N **and**

cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset N*) **and**
atm-incl: *atm-of* ' (*set M*) $\subseteq$ *atms-of-mm N* **and**
distM: *distinct M*
shows
$\exists$ *M' k S*.
lits-of-l M' = *set M* $\wedge$
state S = (*M'*, *N*, {#}, *k*, *None*) $\wedge$
$cdcl_W$*-stgy*** (*init-state N*) *S* $\wedge$
final-$cdcl_W$*-state S*
⟨*proof*⟩

No conflict with only variables of level less than backtrack level

This invariant is stronger than the previous argument in the sense that it is a property about all possible conflicts.

definition *no-smaller-confl* (*S* ::'*st*) $\equiv$
($\forall$ *M K M' D*. *M'* @ *Decided K* # *M* = *trail S* $\longrightarrow$ *D* $\in$# *clauses S*
$\longrightarrow$ $\neg$*M* $\models$*as CNot D*)

lemma *no-smaller-confl-init-sate*[*simp*]:
no-smaller-confl (*init-state N*) ⟨*proof*⟩

lemma $cdcl_W$*-o-no-smaller-confl-inv*:
fixes *S S'* :: '*st*
assumes
$cdcl_W$*-o S S'* **and**
lev: $cdcl_W$*-M-level-inv S* **and**
max-lev: *conflict-is-false-with-level S* **and**
smaller: *no-smaller-confl S* **and**
no-f: *no-clause-is-false S*
shows *no-smaller-confl S'*
⟨*proof*⟩

lemma *conflict-no-smaller-confl-inv*:
assumes *conflict S S'*
and *no-smaller-confl S*
shows *no-smaller-confl S'*
⟨*proof*⟩

lemma *propagate-no-smaller-confl-inv*:
assumes *propagate*: *propagate S S'*
and *n-l*: *no-smaller-confl S*
shows *no-smaller-confl S'*
⟨*proof*⟩

lemma $cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *propagate*: $cdcl_W$*-cp S S'*
and *n-l*: *no-smaller-confl S*
shows *no-smaller-confl S'*
⟨*proof*⟩

lemma *rtrancp-*$cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *propagate*: $cdcl_W$*-cp*** *S S'*
and *n-l*: *no-smaller-confl S*

shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *trancp-cdcl*$_W$*-cp-no-smaller-confl-inv*:
assumes *propagate*: *cdcl*$_W$*-cp*$^{++}$ S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *full-cdcl*$_W$*-cp-no-smaller-confl-inv*:
assumes *full cdcl*$_W$*-cp* S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *full1-cdcl*$_W$*-cp-no-smaller-confl-inv*:
assumes *full1 cdcl*$_W$*-cp* S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-no-smaller-confl-inv*:
assumes *cdcl*$_W$*-stgy* S S'
and *n-l*: *no-smaller-confl* S
and *conflict-is-false-with-level* S
and *cdcl*$_W$*-M-level-inv* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *is-conflicting-exists-conflict*:
assumes $\neg(\forall D{\in}\#\mathit{init\text{-}clss}\ S' + \mathit{learned\text{-}clss}\ S'.\ \neg\ \mathit{trail}\ S' \models\!as\ \mathit{CNot}\ D)$
and *conflicting* $S' = \mathit{None}$
shows $\exists S''.\ \mathit{conflict}\ S'\ S''$
⟨*proof*⟩

lemma *cdcl*$_W$*-o-conflict-is-no-clause-is-false*:
fixes S S' :: *'st*
assumes
cdcl$_W$*-o* S S' **and**
lev: *cdcl*$_W$*-M-level-inv* S **and**
max-lev: *conflict-is-false-with-level* S **and**
no-f: *no-clause-is-false* S **and**
no-l: *no-smaller-confl* S
shows *no-clause-is-false* S'
$\lor$ (*conflicting* $S' = \mathit{None}$
$\longrightarrow (\forall D \in\#\ \mathit{clauses}\ S'.\ \mathit{trail}\ S' \models\!as\ \mathit{CNot}\ D$
$\longrightarrow (\exists L.\ L \in\#\ D \land \mathit{get\text{-}level}\ (\mathit{trail}\ S')\ L = \mathit{backtrack\text{-}lvl}\ S')))$
⟨*proof*⟩

lemma *full1-cdcl*$_W$*-cp-exists-conflict-decompose*:
assumes
confl: $\exists D{\in}\#\mathit{clauses}\ S.\ \mathit{trail}\ S \models\!as\ \mathit{CNot}\ D$ **and**
full: *full cdcl*$_W$*-cp* S U **and**
no-confl: *conflicting* $S = \mathit{None}$ **and**
lev: *cdcl*$_W$*-M-level-inv* S
shows $\exists T.\ \mathit{propagate}^{**}\ S\ T \land \mathit{conflict}\ T\ U$

⟨*proof*⟩

lemma *full1-cdcl$_W$-cp-exists-conflict-full1-decompose*:
assumes
confl: $\exists$ *D*$\in$#*clauses S*. *trail S* $\models$*as CNot D* **and**
full: *full cdcl$_W$-cp S U* **and**
no-confl: *conflicting S* = *None***and**
lev: *cdcl$_W$-M-level-inv S*
shows $\exists$ *T D*. *propagate*** *S T* $\wedge$ *conflict T U*
$\wedge$ *trail T* $\models$*as CNot D* $\wedge$ *conflicting U* = *Some D* $\wedge$ *D* $\in$# *clauses S*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-no-smaller-confl*:
assumes
cdcl$_W$-stgy S S' **and**
n-l: *no-smaller-confl S* **and**
conflict-is-false-with-level S **and**
cdcl$_W$-M-level-inv S **and**
no-clause-is-false S **and**
distinct-cdcl$_W$-state S **and**
cdcl$_W$-conflicting S
shows *no-smaller-confl S'*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-ex-lit-of-max-level*:
assumes
cdcl$_W$-stgy S S' **and**
n-l: *no-smaller-confl S* **and**
conflict-is-false-with-level S **and**
cdcl$_W$-M-level-inv S **and**
no-clause-is-false S **and**
distinct-cdcl$_W$-state S **and**
cdcl$_W$-conflicting S
shows *conflict-is-false-with-level S'*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-stgy-no-smaller-confl-inv*:
assumes
cdcl$_W$-stgy** *S S'* **and**
n-l: *no-smaller-confl S* **and**
cls-false: *conflict-is-false-with-level S* **and**
lev: *cdcl$_W$-M-level-inv S* **and**
no-f: *no-clause-is-false S* **and**
dist: *distinct-cdcl$_W$-state S* **and**
conflicting: *cdcl$_W$-conflicting S* **and**
decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**
learned: *cdcl$_W$-learned-clause S* **and**
alien: *no-strange-atm S*
shows *no-smaller-confl S'* $\wedge$ *conflict-is-false-with-level S'*
⟨*proof*⟩

Final States are Conclusive

lemma *full-cdcl$_W$-stgy-final-state-conclusive-non-false*:
fixes *S'* :: *'st*
assumes *full*: *full cdcl$_W$-stgy* (*init-state N*) *S'*

and *no-d*: *distinct-mset-mset N*
and *no-empty*: $\forall D \in \# N.\ D \neq \{\#\}$
shows (*conflicting* S' = *Some* $\{\#\}$ $\wedge$ *unsatisfiable* (*set-mset* (*init-clss* S')))
$\vee$ (*conflicting* S' = *None* $\wedge$ *trail* S' $\models$*asm* *init-clss* S')
⟨*proof*⟩

lemma *conflict-is-full1-cdcl*$_W$*-cp*:
assumes *cp*: *conflict* S S'
shows *full1 cdcl*$_W$*-cp* S S'
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-fst-empty-conflicting-false*:
assumes
cdcl$_W$*-cp* S S' **and**
trail S = [] **and**
conflicting $S \neq$ *None*
shows *False*
⟨*proof*⟩

lemma *cdcl*$_W$*-o-fst-empty-conflicting-false*:
assumes *cdcl*$_W$*-o* S S'
and *trail* S = []
and *conflicting* $S \neq$ *None*
shows *False*
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-fst-empty-conflicting-false*:
assumes *cdcl*$_W$*-stgy* S S'
and *trail* S = []
and *conflicting* $S \neq$ *None*
shows *False*
⟨*proof*⟩
thm *cdcl*$_W$*-cp.induct*[*split-format*(*complete*)]

lemma *cdcl*$_W$*-cp-conflicting-is-false*:
cdcl$_W$*-cp* S S' $\Longrightarrow$ *conflicting* S = *Some* $\{\#\}$ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-conflicting-is-false*:
cdcl$_W$*-cp*$^{++}$ S S' $\Longrightarrow$ *conflicting* S = *Some* $\{\#\}$ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *cdcl*$_W$*-o-conflicting-is-false*:
cdcl$_W$*-o* S S' $\Longrightarrow$ *conflicting* S = *Some* $\{\#\}$ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-conflicting-is-false*:
cdcl$_W$*-stgy* S S' $\Longrightarrow$ *conflicting* S = *Some* $\{\#\}$ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-conflicting-is-false*:
cdcl$_W$*-stgy*** S S' $\Longrightarrow$ *conflicting* S = *Some* $\{\#\}$ $\Longrightarrow$ $S' = S$
⟨*proof*⟩

lemma *full-cdcl*$_W$*-init-clss-with-false-normal-form*:

assumes
  $\forall\ m \in$ *set* M. $\neg$*is-decided* m **and**
  E = *Some* D **and**
  state S = (M, N, U, *0*, E)
  full $cdcl_W$*-stgy* S S' **and**
  all-decomposition-implies-m (*init-clss* S) (*get-all-ann-decomposition* (*trail* S))
  $cdcl_W$*-learned-clause* S
  $cdcl_W$*-M-level-inv* S
  no-strange-atm S
  distinct-$cdcl_W$*-state* S
  $cdcl_W$*-conflicting* S
shows $\exists M''$. *state* S' = (M'', N, U, *0*, *Some* {#})
⟨*proof*⟩

lemma *full-*$cdcl_W$*-stgy-final-state-conclusive-is-one-false*:
  fixes S' :: *'st*
  assumes *full*: *full* $cdcl_W$*-stgy* (*init-state* N) S'
  and *no-d*: *distinct-mset-mset* N
  and *empty*: {#} $\in$# N
  shows *conflicting* S' = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss* S'))
⟨*proof*⟩

theorem 2.9.9 page 83 of Weidenbach's book

lemma *full-*$cdcl_W$*-stgy-final-state-conclusive*:
  fixes S' :: *'st*
  assumes *full*: *full* $cdcl_W$*-stgy* (*init-state* N) S' **and** *no-d*: *distinct-mset-mset* N
  shows (*conflicting* S' = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss* S')))
    $\vee$ (*conflicting* S' = *None* $\wedge$ *trail* S' $\models$*asm* *init-clss* S')
  ⟨*proof*⟩

theorem 2.9.9 page 83 of Weidenbach's book

lemma *full-*$cdcl_W$*-stgy-final-state-conclusive-from-init-state*:
  fixes S' :: *'st*
  assumes *full*: *full* $cdcl_W$*-stgy* (*init-state* N) S'
  and *no-d*: *distinct-mset-mset* N
  shows (*conflicting* S' = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* N))
   $\vee$ (*conflicting* S' = *None* $\wedge$ *trail* S' $\models$*asm* N $\wedge$ *satisfiable* (*set-mset* N))
⟨*proof*⟩

end
end
theory *CDCL-W-Termination*
imports *CDCL-W*
begin

context *conflict-driven-clause-learning*$_W$
begin

6.1.6 Termination

The condition that no learned clause is a tautology is overkill (in the sense that the no-duplicate condition is enough), but we can reuse *simple-clss*.

The invariant contains all the structural invariants that holds,

definition $cdcl_W$*-all-struct-inv* **where**

$cdcl_W$*-all-struct-inv S* $\longleftrightarrow$
no-strange-atm S $\wedge$
$cdcl_W$*-M-level-inv S* $\wedge$
($\forall$ *s* $\in\#$ *learned-clss S*. $\neg$*tautology s*) $\wedge$
distinct-$cdcl_W$*-state S* $\wedge$
$cdcl_W$*-conflicting S* $\wedge$
all-decomposition-implies-m (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) $\wedge$
$cdcl_W$*-learned-clause S*

lemma $cdcl_W$*-all-struct-inv-inv*:
assumes $cdcl_W$ *S S'* **and** $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-all-struct-inv S'*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-all-struct-inv-inv*:
assumes $cdcl_W{}^{**}$ *S S'* **and** $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-all-struct-inv S'*
⟨*proof*⟩

lemma $cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*:
$cdcl_W$*-stgy S T* $\Longrightarrow$ $cdcl_W$*-all-struct-inv S* $\Longrightarrow$ $cdcl_W$*-all-struct-inv T*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*:
$cdcl_W$*-stgy*** *S T* $\Longrightarrow$ $cdcl_W$*-all-struct-inv S* $\Longrightarrow$ $cdcl_W$*-all-struct-inv T*
⟨*proof*⟩

No Relearning of a clause

lemma $cdcl_W$*-o-new-clause-learned-is-backtrack-step*:
assumes *learned*: *D* $\in\#$ *learned-clss T* **and**
new: *D* $\notin\#$ *learned-clss S* **and**
$cdcl_W$: $cdcl_W$*-o S T* **and**
lev: $cdcl_W$*-M-level-inv S*
shows *backtrack S T* $\wedge$ *conflicting S* = *Some D*
⟨*proof*⟩

lemma $cdcl_W$*-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: *D* $\in\#$ *learned-clss T* **and**
new: *D* $\notin\#$ *learned-clss S* **and**
$cdcl_W$: $cdcl_W$*-stgy S T* **and**
lev: $cdcl_W$*-M-level-inv S*
shows $\exists$ *S'*. *backtrack S S'* $\wedge$ $cdcl_W$*-stgy*** *S' T* $\wedge$ *conflicting S* = *Some D*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: *D* $\in\#$ *learned-clss T* **and**
new: *D* $\notin\#$ *learned-clss S* **and**
$cdcl_W$: $cdcl_W$*-stgy*** *S T* **and**
lev: $cdcl_W$*-M-level-inv S*
shows $\exists$ *S' S''*. $cdcl_W$*-stgy*** *S S'* $\wedge$ *backtrack S' S''* $\wedge$ *conflicting S'* = *Some D* $\wedge$
$cdcl_W$*-stgy*** *S'' T*
⟨*proof*⟩

lemma *propagate-no-more-Decided-lit*:
assumes *propagate S S'*

shows $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$
$\langle proof \rangle$

lemma *conflict-no-more-Decided-lit*:
assumes $conflict\ S\ S'$
shows $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$
$\langle proof \rangle$

lemma $cdcl_W$*-cp-no-more-Decided-lit*:
assumes $cdcl_W\text{-}cp\ S\ S'$
shows $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$
$\langle proof \rangle$

lemma *rtranclp-*$cdcl_W$*-cp-no-more-Decided-lit*:
assumes $cdcl_W\text{-}cp^{**}\ S\ S'$
shows $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$
$\langle proof \rangle$

lemma $cdcl_W$*-o-no-more-Decided-lit*:
assumes $cdcl_W\text{-}o\ S\ S'$ **and** *lev*: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and** $\neg decide\ S\ S'$
shows $Decided\ K \in set\ (trail\ S') \longrightarrow Decided\ K \in set\ (trail\ S)$
$\langle proof \rangle$

lemma $cdcl_W$*-new-decided-at-beginning-is-decide*:
assumes $cdcl_W\text{-}stgy\ S\ S'$ **and**
lev: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and**
$trail\ S' = M'\ @\ Decided\ L\ \#\ M$ **and**
$trail\ S = M$
shows $\exists\, T.\ decide\ S\ T \land no\text{-}step\ cdcl_W\text{-}cp\ S$
$\langle proof \rangle$

lemma $cdcl_W$*-o-is-decide*:
assumes $cdcl_W\text{-}o\ S\ T$ **and** *lev*: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
$trail\ T = drop\ (length\ M_0)\ M'\ @\ Decided\ L\ \#\ H\ @\ M$**and**
$\neg\ (\exists\, M'.\ trail\ S = M'\ @\ Decided\ L\ \#\ H\ @\ M)$
shows $decide\ S\ T$
$\langle proof \rangle$

lemma *rtranclp-*$cdcl_W$*-new-decided-at-beginning-is-decide*:
assumes $cdcl_W\text{-}stgy^{**}\ R\ U$ **and**
$trail\ U = M'\ @\ Decided\ L\ \#\ H\ @\ M$ **and**
$trail\ R = M$ **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ R$
shows
$\exists\, S\ T\ T'.\ cdcl_W\text{-}stgy^{**}\ R\ S \land decide\ S\ T \land cdcl_W\text{-}stgy^{**}\ T\ U \land cdcl_W\text{-}stgy^{**}\ S\ U \land$
$no\text{-}step\ cdcl_W\text{-}cp\ S \land trail\ T = Decided\ L\ \#\ H\ @\ M \land trail\ S = H\ @\ M \land cdcl_W\text{-}stgy\ S\ T' \land$
$cdcl_W\text{-}stgy^{**}\ T'\ U$
$\langle proof \rangle$

lemma *rtranclp-*$cdcl_W$*-new-decided-at-beginning-is-decide'*:
assumes $cdcl_W\text{-}stgy^{**}\ R\ U$ **and**
$trail\ U = M'\ @\ Decided\ L\ \#\ H\ @\ M$ **and**
$trail\ R = M$ **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ R$
shows $\exists\, y\ y'.\ cdcl_W\text{-}stgy^{**}\ R\ y \land cdcl_W\text{-}stgy\ y\ y' \land \neg\ (\exists\, c.\ trail\ y = c\ @\ Decided\ L\ \#\ H\ @\ M)$
$\land\ (\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \land (\exists\, c.\ trail\ a = c\ @\ Decided\ L\ \#\ H\ @\ M))^{**}\ y'\ U$

⟨*proof*⟩

lemma *beginning-not-decided-invert*:
assumes *A*: M @ $A = M'$ @ *Decided* K # H **and**
nm: $\forall m \in set\ M.\ \neg$*is-decided* m
shows $\exists M.\ A = M$ @ *Decided* K # H
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-trail-has-new-decided-is-decide-step*:
assumes *cdcl*$_W$*-stgy* S T
$\neg$ ($\exists c.$ *trail* $S = c$ @ *Decided* L # H @ M) **and**
($\lambda a\ b.$ *cdcl*$_W$*-stgy* $a\ b \wedge$ ($\exists c.$ *trail* $a = c$ @ *Decided* L # H @ M))** T U **and**
$\exists M'.$ *trail* $U = M'$ @ *Decided* L # H @ M **and**
lev: *cdcl*$_W$*-M-level-inv* S
shows $\exists S'.$ *decide* S $S' \wedge$ *full cdcl*$_W$*-cp* S' $T \wedge$ *no-step cdcl*$_W$*-cp* S
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-with-trail-end-has-trail-end*:
assumes ($\lambda a\ b.$ *cdcl*$_W$*-stgy* $a\ b \wedge$ ($\exists c.$ *trail* $a = c$ @ *Decided* L # H @ M))** T U **and**
$\exists M'.$ *trail* $U = M'$ @ *Decided* L # H @ M
shows $\exists M'.$ *trail* $T = M'$ @ *Decided* L # H @ M
⟨*proof*⟩

lemma *remove1-mset-eq-remove1-mset-same*:
remove1-mset L D = *remove1-mset* L' $D \Longrightarrow L \in\# D \Longrightarrow L = L'$
⟨*proof*⟩

lemma *cdcl*$_W$*-o-cannot-learn*:
assumes
cdcl$_W$*-o* y z **and**
lev: *cdcl*$_W$*-M-level-inv* y **and**
M: *trail* $y = c$ @ *Decided* Kh # H **and**
DL: $D \notin\#$ *learned-clss* y **and**
LD: $L \in\# D$ **and**
DH: *atms-of* (*remove1-mset* L D) $\subseteq$ *atm-of* ‘ *lits-of-l* H **and**
LH: *atm-of* $L \notin$ *atm-of* ‘ *lits-of-l* H **and**
learned: $\forall T.$ *conflicting* y = *Some* $T \longrightarrow$ *trail* $y \models$*as CNot* T **and**
z: *trail* $z = c'$ @ *Decided* Kh # H
shows $D \notin\#$ *learned-clss* z
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-with-trail-end-has-not-been-learned*:
assumes
cdcl$_W$*-stgy* y z **and**
cdcl$_W$*-M-level-inv* y **and**
trail $y = c$ @ *Decided* Kh # H **and**
$D \notin\#$ *learned-clss* y **and**
LD: $L \in\# D$ **and**
DH: *atms-of* (*remove1-mset* L D) $\subseteq$ *atm-of* ‘ *lits-of-l* H **and**
LH: *atm-of* $L \notin$ *atm-of* ‘ *lits-of-l* H **and**
$\forall T.$ *conflicting* y = *Some* $T \longrightarrow$ *trail* $y \models$*as CNot* T **and**
trail $z = c'$ @ *Decided* Kh # H
shows $D \notin\#$ *learned-clss* z
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-with-trail-end-has-not-been-learned*:

assumes
$(\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \wedge (\exists c.\ trail\ a = c\ @\ Decided\ K\#\ H\ @\ []))^{**}\ S\ z$ **and**
$cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ **and**
$trail\ S = c\ @\ Decided\ K\ \#\ H$ **and**
$D \notin\#\ learned\text{-}clss\ S$ **and**
LD: $L \in\#\ D$ **and**
DH: $atms\text{-}of\ (remove1\text{-}mset\ L\ D) \subseteq atm\text{-}of\ `\ lits\text{-}of\text{-}l\ H$ **and**
LH: $atm\text{-}of\ L \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ H$ **and**
$\exists c'.\ trail\ z = c'\ @\ Decided\ K\ \#\ H$
shows $D \notin\#\ learned\text{-}clss\ z$
⟨*proof*⟩

lemma $cdcl_W$*-stgy-new-learned-clause*:
assumes $cdcl_W\text{-}stgy\ S\ T$ **and**
lev: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and**
$E \notin\#\ learned\text{-}clss\ S$ **and**
$E \in\#\ learned\text{-}clss\ T$
shows $\exists S'.\ backtrack\ S\ S' \wedge conflicting\ S = Some\ E \wedge full\ cdcl_W\text{-}cp\ S'\ T$
⟨*proof*⟩

theorem 2.9.7 page 83 of Weidenbach's book

lemma $cdcl_W$*-stgy-no-relearned-clause*:
assumes
invR: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$ **and**
st': $cdcl_W\text{-}stgy^{**}\ R\ S$ **and**
bt: $backtrack\ S\ T$ **and**
confl: $conflicting\ S = Some\ E$ **and**
already-learned: $E \in\#\ clauses\ S$ **and**
R: $trail\ R = []$
shows *False*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-stgy-distinct-mset-clauses*:
assumes
invR: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$ **and**
st: $cdcl_W\text{-}stgy^{**}\ R\ S$ **and**
dist: $distinct\text{-}mset\ (clauses\ R)$ **and**
R: $trail\ R = []$
shows $distinct\text{-}mset\ (clauses\ S)$
⟨*proof*⟩

lemma $cdcl_W$*-stgy-distinct-mset-clauses*:
assumes
st: $cdcl_W\text{-}stgy^{**}\ (init\text{-}state\ N)\ S$ **and**
no-duplicate-clause: $distinct\text{-}mset\ N$ **and**
no-duplicate-in-clause: $distinct\text{-}mset\text{-}mset\ N$
shows $distinct\text{-}mset\ (clauses\ S)$
⟨*proof*⟩

Decrease of a Measure

fun $cdcl_W$*-measure* **where**
$cdcl_W\text{-}measure\ S =$
$[(3::nat)\ \hat{}\ (card\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S))) - card\ (set\text{-}mset\ (learned\text{-}clss\ S)),$
$if\ conflicting\ S = None\ then\ 1\ else\ 0,$
$if\ conflicting\ S = None\ then\ card\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S)) - length\ (trail\ S)$

else length (*trail S*)
]

lemma *length-model-le-vars-all-inv*:
assumes $cdcl_W$*-all-struct-inv S*
shows *length* (*trail S*) $\leq$ *card* (*atms-of-mm* (*init-clss S*))
⟨*proof*⟩
end

context *conflict-driven-clause-learning*$_W$
begin

lemma *learned-clss-less-upper-bound*:
fixes *S* :: *'st*
assumes
distinct-cdcl$_W$*-state S* **and**
$\forall$ *s* $\in\#$ *learned-clss S*. $\neg$*tautology s*
shows *card*(*set-mset* (*learned-clss S*)) $\leq$ *3* ^ *card* (*atms-of-mm* (*learned-clss S*))
⟨*proof*⟩

lemma $cdcl_W$*-measure-decreasing*:
fixes *S* :: *'st*
assumes
$cdcl_W$ *S S'* **and**
no-restart:
$\neg$(*learned-clss S* $\subseteq\#$ *learned-clss S'* $\wedge$ [] = *trail S'* $\wedge$ *conflicting S'* = *None*)
and
no-forget: *learned-clss S* $\subseteq\#$ *learned-clss S'* **and**
no-relearn: $\bigwedge$*S'*. *backtrack S S'* $\Longrightarrow$ $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *T* $\notin\#$ *learned-clss S*
and
alien: *no-strange-atm S* **and**
M-level: $cdcl_W$*-M-level-inv S* **and**
no-taut: $\forall$ *s* $\in\#$ *learned-clss S*. $\neg$*tautology s* **and**
no-dup: *distinct-cdcl*$_W$*-state S* **and**
confl: $cdcl_W$*-conflicting S*
shows ($cdcl_W$*-measure S'*, $cdcl_W$*-measure S*) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *propagate-measure-decreasing*:
fixes *S* :: *'st*
assumes *propagate S S'* **and** $cdcl_W$*-all-struct-inv S*
shows ($cdcl_W$*-measure S'*, $cdcl_W$*-measure S*) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *conflict-measure-decreasing*:
fixes *S* :: *'st*
assumes *conflict S S'* **and** $cdcl_W$*-all-struct-inv S*
shows ($cdcl_W$*-measure S'*, $cdcl_W$*-measure S*) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *decide-measure-decreasing*:
fixes *S* :: *'st*
assumes *decide S S'* **and** $cdcl_W$*-all-struct-inv S*
shows ($cdcl_W$*-measure S'*, $cdcl_W$*-measure S*) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *cdcl$_W$-cp-measure-decreasing*:
fixes S :: *'st*
assumes *cdcl$_W$-cp* S S' **and** *cdcl$_W$-all-struct-inv* S
shows (*cdcl$_W$-measure* S', *cdcl$_W$-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-measure-decreasing*:
fixes S :: *'st*
assumes *cdcl$_W$-cp^{++}* S S' **and** *cdcl$_W$-all-struct-inv* S
shows (*cdcl$_W$-measure* S', *cdcl$_W$-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-step-decreasing*:
fixes R S T :: *'st*
assumes *cdcl$_W$-stgy* S T **and**
cdcl$_W$-stgy** R S
trail R = [] **and**
cdcl$_W$-all-struct-inv R
shows (*cdcl$_W$-measure* T, *cdcl$_W$-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

Roughly corresponds to theorem 2.9.15 page 86 of Weidenbach's book (using a different bound)

lemma *tranclp-cdcl$_W$-stgy-decreasing*:
fixes R S T :: *'st*
assumes *cdcl$_W$-stgy^{++}* R S
trail R = [] **and**
cdcl$_W$-all-struct-inv R
shows (*cdcl$_W$-measure* S, *cdcl$_W$-measure* R) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-stgy-S0-decreasing*:
fixes R S T :: *'st*
assumes
pl: *cdcl$_W$-stgy^{++}* (*init-state* N) S **and**
no-dup: *distinct-mset-mset* N
shows (*cdcl$_W$-measure* S, *cdcl$_W$-measure* (*init-state* N)) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *wf-tranclp-cdcl$_W$-stgy*:
wf {(S::*'st*, *init-state* N)|
S N. *distinct-mset-mset* N $\wedge$ *cdcl$_W$-stgy^{++}* (*init-state* N) S}
⟨*proof*⟩

lemma *cdcl$_W$-cp-wf-all-inv*:
wf {(S', S). *cdcl$_W$-all-struct-inv* S $\wedge$ *cdcl$_W$-cp* S S'}
(**is** *wf ?R*)
⟨*proof*⟩

end

end

6.2 Merging backjump rules

theory *CDCL-W-Merge*
imports *CDCL-W-Termination*
begin

Before showing that Weidenbach's CDCL is included in NOT's CDCL, we need to work on a variant of Weidenbach's calculus: NOT's backjump assumes the existence of a clause that is suitable to backjump. This clause is obtained in W's CDCL by applying:

1. $\textit{conflict-driven-clause-learning}_W\textit{.conflict}$ to find the conflict
2. the conflict is analysed by repetitive application of $\textit{conflict-driven-clause-learning}_W\textit{.resolve}$ and $\textit{conflict-driven-clause-learning}_W\textit{.skip}$,
3. finally $\textit{conflict-driven-clause-learning}_W\textit{.backtrack}$ is used to backtrack.

We show that this new calculus has the same final states than Weidenbach's CDCL if the calculus starts in a state such that the invariant holds and no conflict has been found yet. The latter condition holds for initial states.

6.2.1 Inclusion of the states

context $\textit{conflict-driven-clause-learning}_W$
begin
declare $\textit{cdcl}_W\textit{.intros}[\textit{intro}]$ $\textit{cdcl}_W\textit{-bj.intros}[\textit{intro}]$ $\textit{cdcl}_W\textit{-o.intros}[\textit{intro}]$

lemma $\textit{backtrack-no-cdcl}_W\textit{-bj}$:
  assumes *cdcl*: $\textit{cdcl}_W\textit{-bj}\ T\ U$ **and** *inv*: $\textit{cdcl}_W\textit{-M-level-inv}\ V$
  shows $\neg\textit{backtrack}\ V\ T$
  ⟨*proof*⟩

skip-or-resolve corresponds to the *analyze* function in the code of MiniSAT.

inductive *skip-or-resolve* :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$\textit{s-or-r-skip}[\textit{intro}]$: $\textit{skip}\ S\ T \Longrightarrow \textit{skip-or-resolve}\ S\ T$ |
$\textit{s-or-r-resolve}[\textit{intro}]$: $\textit{resolve}\ S\ T \Longrightarrow \textit{skip-or-resolve}\ S\ T$

lemma $\textit{rtranclp-cdcl}_W\textit{-bj-skip-or-resolve-backtrack}$:
  assumes $\textit{cdcl}_W\textit{-bj}^{**}\ S\ U$ **and** *inv*: $\textit{cdcl}_W\textit{-M-level-inv}\ S$
  shows $\textit{skip-or-resolve}^{**}\ S\ U \lor (\exists\, T.\ \textit{skip-or-resolve}^{**}\ S\ T \land \textit{backtrack}\ T\ U)$
  ⟨*proof*⟩

lemma $\textit{rtranclp-skip-or-resolve-rtranclp-cdcl}_W$:
  $\textit{skip-or-resolve}^{**}\ S\ T \Longrightarrow \textit{cdcl}_W{}^{**}\ S\ T$
  ⟨*proof*⟩

definition *backjump-l-cond* :: $'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **where**
$\textit{backjump-l-cond} \equiv \lambda C\ C'\ L'\ S\ T.\ \textit{True}$

definition $\textit{inv}_{NOT}$ :: $'st \Rightarrow bool$ **where**
$\textit{inv}_{NOT} \equiv \lambda S.\ \textit{no-dup}\ (\textit{trail}\ S)$

declare $\textit{inv}_{NOT}\textit{-def}[\textit{simp}]$
end

context *conflict-driven-clause-learning*$_W$
begin

6.2.2 More lemmas conflict–propagate and backjumping

Termination

lemma *cdcl*$_W$*-cp-normalized-element-all-inv*:
  assumes *inv*: *cdcl*$_W$*-all-struct-inv S*
  obtains *T* **where** *full cdcl*$_W$*-cp S T*
  ⟨*proof*⟩
thm *backtrackE*

lemma *cdcl*$_W$*-bj-measure*:
  assumes *cdcl*$_W$*-bj S T* **and** *cdcl*$_W$*-M-level-inv S*
  shows *length* (*trail S*) + (*if conflicting S* = *None then 0 else 1*)
    > *length* (*trail T*) + (*if conflicting T* = *None then 0 else 1*)
  ⟨*proof*⟩

lemma *wf-cdcl*$_W$*-bj*:
  wf {(*b*,*a*). *cdcl*$_W$*-bj a b* ∧ *cdcl*$_W$*-M-level-inv a*}
  ⟨*proof*⟩

lemma *cdcl*$_W$*-bj-exists-normal-form*:
  assumes *lev*: *cdcl*$_W$*-M-level-inv S*
  shows ∃ *T*. *full cdcl*$_W$*-bj S T*
⟨*proof*⟩
lemma *rtranclp-skip-state-decomp*:
  assumes *skip*** *S T* **and** *no-dup* (*trail S*)
  shows
    ∃ *M*. *trail S* = *M* @ *trail T* ∧ (∀ *m*∈*set M*. ¬*is-decided m*)
    init-clss S = *init-clss T*
    learned-clss S = *learned-clss T*
    backtrack-lvl S = *backtrack-lvl T*
    conflicting S = *conflicting T*
  ⟨*proof*⟩

More backjumping

Backjumping after skipping or jump directly **lemma** *rtranclp-skip-backtrack-backtrack*:
  assumes
    skip** *S T* **and**
    backtrack T W **and**
    cdcl$_W$*-all-struct-inv S*
  shows *backtrack S W*
  ⟨*proof*⟩

See also theorem *rtranclp-skip-backtrack-backtrack*

lemma *rtranclp-skip-backtrack-backtrack-end*:
  assumes
    skip: *skip*** *S T* **and**
    bt: *backtrack S W* **and**
    inv: *cdcl*$_W$*-all-struct-inv S*
  shows *backtrack T W*
  ⟨*proof*⟩

lemma *$cdcl_W$-bj-decomp-resolve-skip-and-bj*:
assumes *$cdcl_W$-bj** S T* **and** *inv*: *$cdcl_W$-M-level-inv S*
shows (*skip-or-resolve** S T*
∨ (∃ *U*. *skip-or-resolve** S U* ∧ *backtrack U T*))
⟨*proof*⟩

lemma *resolve-skip-deterministic*:
resolve S T ⟹ *skip S U* ⟹ *False*
⟨*proof*⟩

lemma *list-same-level-decomp-is-same-decomp*:
assumes *M-K*: *M* = *M1* @ *Decided K* # *M2* **and** *M-K'*: *M* = *M1'* @ *Decided K'* # *M2'* **and**
lev-KK': *get-level M K* = *get-level M K'* **and**
n-d: *no-dup M*
shows *K* = *K'* **and** *M1* = *M1'* **and** *M2* = *M2'*
⟨*proof*⟩

lemma *backtrack-unique*:
assumes
bt-T: *backtrack S T* **and**
bt-U: *backtrack S U* **and**
inv: *$cdcl_W$-all-struct-inv S*
shows *T* ∼ *U*
⟨*proof*⟩

lemma *if-can-apply-backtrack-no-more-resolve*:
assumes
skip: *skip** S U* **and**
bt: *backtrack S T* **and**
inv: *$cdcl_W$-all-struct-inv S*
shows ¬*resolve U V*
⟨*proof*⟩

lemma *if-can-apply-resolve-no-more-backtrack*:
assumes
skip: *skip** S U* **and**
resolve: *resolve S T* **and**
inv: *$cdcl_W$-all-struct-inv S*
shows ¬*backtrack U V*
⟨*proof*⟩

lemma *if-can-apply-backtrack-skip-or-resolve-is-skip*:
assumes
bt: *backtrack S T* **and**
skip: *skip-or-resolve** S U* **and**
inv: *$cdcl_W$-all-struct-inv S*
shows *skip** S U*
⟨*proof*⟩

lemma *$cdcl_W$-bj-bj-decomp*:
assumes *$cdcl_W$-bj** S W* **and** *$cdcl_W$-all-struct-inv S*
shows
(∃ *T U V*. (*λS T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T*
∧ (*λT U*. *resolve T U* ∧ *no-step backtrack T*) *T U*
∧ *skip** U V* ∧ *backtrack V W*)
∨ (∃ *T U*. (*λS T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T*

$\wedge$ (λT U. *resolve* T U $\wedge$ *no-step backtrack* T) T U $\wedge$ *skip*** U W)
$\vee$ ($\exists$ T. *skip*** S T $\wedge$ *backtrack* T W)
$\vee$ *skip*** S W (**is** *?RB* S W $\vee$ *?R* S W $\vee$ *?SB* S W $\vee$ *?S* S W)
⟨*proof*⟩

The case distinction is needed, since $T \sim V$ does not imply that R^{**} T V.

lemma *cdcl*$_W$*-bj-strongly-confluent*:
assumes
cdcl$_W$*-bj*** S V **and**
cdcl$_W$*-bj*** S T **and**
n-s: *no-step cdcl*$_W$*-bj* V **and**
inv: *cdcl*$_W$*-all-struct-inv* S
shows $T \sim V$ $\vee$ *cdcl*$_W$*-bj*** T V
⟨*proof*⟩

lemma *cdcl*$_W$*-bj-unique-normal-form*:
assumes
ST: *cdcl*$_W$*-bj*** S T **and** *SU*: *cdcl*$_W$*-bj*** S U **and**
n-s-U: *no-step cdcl*$_W$*-bj* U **and**
n-s-T: *no-step cdcl*$_W$*-bj* T **and**
inv: *cdcl*$_W$*-all-struct-inv* S
shows $T \sim U$
⟨*proof*⟩

lemma *full-cdcl*$_W$*-bj-unique-normal-form*:
assumes *full cdcl*$_W$*-bj* S T **and** *full cdcl*$_W$*-bj* S U **and**
inv: *cdcl*$_W$*-all-struct-inv* S
shows $T \sim U$
⟨*proof*⟩

6.2.3 CDCL with Merging

inductive *cdcl*$_W$*-merge-restart* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **where**
fw-r-propagate: *propagate* S S' $\Longrightarrow$ *cdcl*$_W$*-merge-restart* S S' |
fw-r-conflict: *conflict* S T $\Longrightarrow$ *full cdcl*$_W$*-bj* T U $\Longrightarrow$ *cdcl*$_W$*-merge-restart* S U |
fw-r-decide: *decide* S S' $\Longrightarrow$ *cdcl*$_W$*-merge-restart* S S'|
fw-r-rf: *cdcl*$_W$*-rf* S S' $\Longrightarrow$ *cdcl*$_W$*-merge-restart* S S'

lemma *rtranclp-cdcl*$_W$*-bj-rtranclp-cdcl*$_W$:
cdcl$_W$*-bj*** S T $\Longrightarrow$ *cdcl*$_W$** S T
⟨*proof*⟩

lemma *cdcl*$_W$*-merge-restart-cdcl*$_W$:
assumes *cdcl*$_W$*-merge-restart* S T
shows *cdcl*$_W$** S T
⟨*proof*⟩

lemma *cdcl*$_W$*-merge-restart-conflicting-true-or-no-step*:
assumes *cdcl*$_W$*-merge-restart* S T
shows *conflicting* T = *None* $\vee$ *no-step cdcl*$_W$ T
⟨*proof*⟩

inductive *cdcl*$_W$*-merge* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **where**
fw-propagate: *propagate* S S' $\Longrightarrow$ *cdcl*$_W$*-merge* S S' |
fw-conflict: *conflict* S T $\Longrightarrow$ *full cdcl*$_W$*-bj* T U $\Longrightarrow$ *cdcl*$_W$*-merge* S U |

fw-decide: *decide* S S' $\Longrightarrow$ $cdcl_W$*-merge* S S'|
fw-forget: *forget* S S' $\Longrightarrow$ $cdcl_W$*-merge* S S'

lemma $cdcl_W$*-merge-*$cdcl_W$*-merge-restart*:
$cdcl_W$*-merge* S T $\Longrightarrow$ $cdcl_W$*-merge-restart* S T
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-tranclp-*$cdcl_W$*-merge-restart*:
$cdcl_W$*-merge*** S T $\Longrightarrow$ $cdcl_W$*-merge-restart*** S T
⟨*proof*⟩

lemma $cdcl_W$*-merge-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge* S T $\Longrightarrow$ $cdcl_W{}^{**}$ S T
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge*** S T $\Longrightarrow$ $cdcl_W{}^{**}$ S T
⟨*proof*⟩

lemmas *rulesE* =
skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE

lemma $cdcl_W$*-all-struct-inv-tranclp-*$cdcl_W$*-merge-tranclp-*$cdcl_W$*-merge-*$cdcl_W$*-all-struct-inv*:
assumes
inv: $cdcl_W$*-all-struct-inv* b
$cdcl_W$*-merge*$^{++}$ b a
shows $(\lambda S\ T.\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \land cdcl_W\text{-}merge\ S\ T)^{++}\ b\ a$
⟨*proof*⟩

lemma *backtrack-is-full1-*$cdcl_W$*-bj*:
assumes *bt*: *backtrack* S T **and** *inv*: $cdcl_W$*-M-level-inv* S
shows *full1* $cdcl_W$*-bj* S T
⟨*proof*⟩

lemma *rtrancl-*$cdcl_W$*-conflicting-true-*$cdcl_W$*-merge-restart*:
assumes $cdcl_W{}^{**}$ S V **and** *inv*: $cdcl_W$*-M-level-inv* S **and** *conflicting* S = *None*
shows ($cdcl_W$*-merge-restart*** S V $\land$ *conflicting* V = *None*)
$\lor$ ($\exists$ T U. $cdcl_W$*-merge-restart*** S T $\land$ *conflicting* $V \neq$ *None* $\land$ *conflict* T U $\land$ $cdcl_W$*-bj*** U V)
⟨*proof*⟩

lemma *no-step-*$cdcl_W$*-no-step-*$cdcl_W$*-merge-restart*: *no-step* $cdcl_W$ S $\Longrightarrow$ *no-step* $cdcl_W$*-merge-restart* S
⟨*proof*⟩

lemma *no-step-*$cdcl_W$*-merge-restart-no-step-*$cdcl_W$:
assumes
conflicting S = *None* **and**
$cdcl_W$*-M-level-inv* S **and**
no-step $cdcl_W$*-merge-restart* S
shows *no-step* $cdcl_W$ S
⟨*proof*⟩

lemma $cdcl_W$*-merge-restart-no-step-*$cdcl_W$*-bj*:
assumes
$cdcl_W$*-merge-restart* S T
shows *no-step* $cdcl_W$*-bj* T

$\langle proof \rangle$

lemma *rtranclp-cdcl$_W$-merge-restart-no-step-cdcl$_W$-bj*:
assumes
cdcl$_W$-merge-restart** S T **and**
conflicting S = None
shows *no-step cdcl$_W$-bj T*
$\langle proof \rangle$

If *conflicting S ≠ None*, we cannot say anything.

Remark that this theorem does not say anything about well-foundedness: even if you know that one relation is well-founded, it only states that the normal forms are shared.

lemma *conflicting-true-full-cdcl$_W$-iff-full-cdcl$_W$-merge*:
assumes *confl*: *conflicting S = None* **and** *lev*: *cdcl$_W$-M-level-inv S*
shows *full cdcl$_W$ S V $\longleftrightarrow$ full cdcl$_W$-merge-restart S V*
$\langle proof \rangle$

lemma *init-state-true-full-cdcl$_W$-iff-full-cdcl$_W$-merge*:
shows *full cdcl$_W$ (init-state N) V $\longleftrightarrow$ full cdcl$_W$-merge-restart (init-state N) V*
$\langle proof \rangle$

6.2.4 CDCL with Merge and Strategy

The intermediate step

inductive *cdcl$_W$-s'* :: *'st ⇒ 'st ⇒ bool* **where**
conflict': *full1 cdcl$_W$-cp S S' $\Longrightarrow$ cdcl$_W$-s' S S'* |
decide': *decide S S' $\Longrightarrow$ no-step cdcl$_W$-cp S $\Longrightarrow$ full cdcl$_W$-cp S' S'' $\Longrightarrow$ cdcl$_W$-s' S S''* |
bj': *full1 cdcl$_W$-bj S S' $\Longrightarrow$ no-step cdcl$_W$-cp S $\Longrightarrow$ full cdcl$_W$-cp S' S'' $\Longrightarrow$ cdcl$_W$-s' S S''*

inductive-cases *cdcl$_W$-s'E*: *cdcl$_W$-s' S T*

lemma *rtranclp-cdcl$_W$-bj-full1-cdclp-cdcl$_W$-stgy*:
cdcl$_W$-bj** S S' $\Longrightarrow$ full cdcl$_W$-cp S' S'' $\Longrightarrow$ cdcl$_W$-stgy** S S''
$\langle proof \rangle$

lemma *cdcl$_W$-s'-is-rtranclp-cdcl$_W$-stgy*:
cdcl$_W$-s' S T $\Longrightarrow$ cdcl$_W$-stgy** S T
$\langle proof \rangle$

lemma *cdcl$_W$-cp-cdcl$_W$-bj-bissimulation*:
assumes
full cdcl$_W$-cp T U **and**
cdcl$_W$-bj** T T' **and**
cdcl$_W$-all-struct-inv T **and**
no-step cdcl$_W$-bj T'
shows *full cdcl$_W$-cp T' U*
∨ (∃ *U' U''*. *full cdcl$_W$-cp T' U''* ∧ *full1 cdcl$_W$-bj U U'* ∧ *full cdcl$_W$-cp U' U''*
∧ *cdcl$_W$-s'** U U''*)
$\langle proof \rangle$

lemma *cdcl$_W$-cp-cdcl$_W$-bj-bissimulation'*:
assumes
full cdcl$_W$-cp T U **and**
cdcl$_W$-bj** T T' **and**
cdcl$_W$-all-struct-inv T **and**

no-step $cdcl_W$*-bj* T'
shows *full* $cdcl_W$*-cp* T' U
$\lor$ ($\exists$ U'. *full1* $cdcl_W$*-bj* U U' $\land$ ($\forall$ U''. *full* $cdcl_W$*-cp* U' U'' $\longrightarrow$ *full* $cdcl_W$*-cp* T' U''
$\land$ $cdcl_W$*-s'*** U U''))
⟨*proof*⟩

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s'-connected*:
assumes $cdcl_W$*-stgy* S U **and** $cdcl_W$*-all-struct-inv* S
shows $cdcl_W$*-s'* S U
$\lor$ ($\exists$ U'. *full1* $cdcl_W$*-bj* U U' $\land$ ($\forall$ U''. *full* $cdcl_W$*-cp* U' U'' $\longrightarrow$ $cdcl_W$*-s'* S U''))
⟨*proof*⟩

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s'-connected'*:
assumes $cdcl_W$*-stgy* S U **and** $cdcl_W$*-all-struct-inv* S
shows $cdcl_W$*-s'* S U
$\lor$ ($\exists$ U' U''. $cdcl_W$*-s'* S U'' $\land$ *full1* $cdcl_W$*-bj* U U' $\land$ *full* $cdcl_W$*-cp* U' U'')
⟨*proof*⟩

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s'-no-step*:
assumes $cdcl_W$*-stgy* S U **and** $cdcl_W$*-all-struct-inv* S **and** *no-step* $cdcl_W$*-bj* U
shows $cdcl_W$*-s'* S U
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-stgy-connected-to-rtranclp-*$cdcl_W$*-s'*:
assumes $cdcl_W$*-stgy*** S U **and** *inv*: $cdcl_W$*-M-level-inv* S
shows $cdcl_W$*-s'*** S U $\lor$ ($\exists$ T. $cdcl_W$*-s'*** S T $\land$ $cdcl_W$*-bj*$^{++}$ T U $\land$ *conflicting* $U \neq$ *None*)
⟨*proof*⟩

lemma *n-step-*$cdcl_W$*-stgy-iff-no-step-*$cdcl_W$*-cl-*$cdcl_W$*-o*:
assumes *inv*: $cdcl_W$*-all-struct-inv* S
shows *no-step* $cdcl_W$*-s'* S $\longleftrightarrow$ *no-step* $cdcl_W$*-cp* S $\land$ *no-step* $cdcl_W$*-o* S (**is** *?S'* S $\longleftrightarrow$ *?C* S $\land$ *?O* S)
⟨*proof*⟩

lemma $cdcl_W$*-s'-tranclp-*$cdcl_W$:
$cdcl_W$*-s'* S S' $\Longrightarrow$ $cdcl_W{}^{++}$ S S'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-s'-tranclp-*$cdcl_W$:
$cdcl_W$*-s'*$^{++}$ S S' $\Longrightarrow$ $cdcl_W{}^{++}$ S S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-s'-rtranclp-*$cdcl_W$:
$cdcl_W$*-s'*** S S' $\Longrightarrow$ $cdcl_W{}^{**}$ S S'
⟨*proof*⟩

lemma *full-*$cdcl_W$*-stgy-iff-full-*$cdcl_W$*-s'*:
assumes *inv*: $cdcl_W$*-all-struct-inv* S
shows *full* $cdcl_W$*-stgy* S T $\longleftrightarrow$ *full* $cdcl_W$*-s'* S T (**is** *?S* $\longleftrightarrow$ *?S'*)
⟨*proof*⟩

lemma *conflict-step-*$cdcl_W$*-stgy-step*:
assumes
conflict S T
$cdcl_W$*-all-struct-inv* S
shows $\exists$ T. $cdcl_W$*-stgy* S T
⟨*proof*⟩

lemma *decide-step-cdcl$_W$-stgy-step*:
  assumes
    decide S T
    cdcl$_W$-all-struct-inv S
  shows $\exists$ *T*. *cdcl$_W$-stgy S T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-conflicting-Some*:
  cdcl$_W$-cp** S T $\Longrightarrow$ *conflicting S = Some D* $\Longrightarrow$ *S = T*
  ⟨*proof*⟩

inductive *cdcl$_W$-merge-cp* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
conflict': *conflict S T* $\Longrightarrow$ *full cdcl$_W$-bj T U* $\Longrightarrow$ *cdcl$_W$-merge-cp S U* |
propagate': *propagate^{++} S S'* $\Longrightarrow$ *cdcl$_W$-merge-cp S S'*

lemma *cdcl$_W$-merge-restart-cases*[*consumes 1*, *case-names conflict propagate*]:
  assumes
    cdcl$_W$-merge-cp S U **and**
    $\bigwedge$*T*. *conflict S T* $\Longrightarrow$ *full cdcl$_W$-bj T U* $\Longrightarrow$ *P* **and**
    propagate^{++} S U $\Longrightarrow$ *P*
  shows *P*
  ⟨*proof*⟩

lemma *cdcl$_W$-merge-cp-tranclp-cdcl$_W$-merge*:
  cdcl$_W$-merge-cp S T $\Longrightarrow$ *cdcl$_W$-merge^{++} S T*
  ⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-cp-rtranclp-cdcl$_W$*:
  cdcl$_W$-merge-cp** S T $\Longrightarrow$ *cdcl$_W$** S T*
 ⟨*proof*⟩

lemma *full1-cdcl$_W$-bj-no-step-cdcl$_W$-bj*:
  full1 cdcl$_W$-bj S T $\Longrightarrow$ *no-step cdcl$_W$-cp S*
  ⟨*proof*⟩

Full Transformation

inductive *cdcl$_W$-s'-without-decide* **where**
conflict'-without-decide[*intro*]: *full1 cdcl$_W$-cp S S'* $\Longrightarrow$ *cdcl$_W$-s'-without-decide S S'* |
bj'-without-decide[*intro*]: *full1 cdcl$_W$-bj S S'* $\Longrightarrow$ *no-step cdcl$_W$-cp S* $\Longrightarrow$ *full cdcl$_W$-cp S' S''*
    $\Longrightarrow$ *cdcl$_W$-s'-without-decide S S''*

lemma *rtranclp-cdcl$_W$-s'-without-decide-rtranclp-cdcl$_W$*:
  cdcl$_W$-s'-without-decide** S T $\Longrightarrow$ *cdcl$_W$** S T*
  ⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s'-without-decide-rtranclp-cdcl$_W$-s'*:
  cdcl$_W$-s'-without-decide** S T $\Longrightarrow$ *cdcl$_W$-s'** S T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-cp-is-rtranclp-cdcl$_W$-s'-without-decide*:
  assumes
    cdcl$_W$-merge-cp** S V
    conflicting S = None
  shows

$(cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ V)$
$\lor\ (\exists\, T.\ cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ T \land propagate^{++}\ T\ V)$
$\lor\ (\exists\, T\ U.\ cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ T \land full1\ cdcl_W\text{-}bj\ T\ U \land propagate^{**}\ U\ V)$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-s'-without-decide-is-rtranclp-cdcl*$_W$*-merge-cp*:
assumes
$cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ V$ **and**
confl: $conflicting\ S = None$
shows
$(cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V \land conflicting\ V = None)$
$\lor\ (cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V \land conflicting\ V \neq None \land no\text{-}step\ cdcl_W\text{-}cp\ V \land no\text{-}step\ cdcl_W\text{-}bj\ V)$
$\lor\ (\exists\, T.\ cdcl_W\text{-}merge\text{-}cp^{**}\ S\ T \land conflict\ T\ V)$
⟨*proof*⟩

lemma *no-step-cdcl*$_W$*-s'-no-ste-cdcl*$_W$*-merge-cp*:
assumes
$cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
$conflicting\ S = None$
$no\text{-}step\ cdcl_W\text{-}s'\ S$
shows $no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S$
⟨*proof*⟩

The *no-step decide S* is needed, since *cdcl*$_W$*-merge-cp* is *cdcl*$_W$*-s'* without *decide*.

lemma *conflicting-true-no-step-cdcl*$_W$*-merge-cp-no-step-s'-without-decide*:
assumes
confl: $conflicting\ S = None$ **and**
inv: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and**
n-s: $no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S$
shows $no\text{-}step\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S$
⟨*proof*⟩

lemma *conflicting-true-no-step-s'-without-decide-no-step-cdcl*$_W$*-merge-cp*:
assumes
inv: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ **and**
n-s: $no\text{-}step\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S$
shows $no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S$
⟨*proof*⟩

lemma *no-step-cdcl*$_W$*-merge-cp-no-step-cdcl*$_W$*-cp*:
$no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S \Longrightarrow cdcl_W\text{-}M\text{-}level\text{-}inv\ S \Longrightarrow no\text{-}step\ cdcl_W\text{-}cp\ S$
⟨*proof*⟩

lemma *conflicting-not-true-rtranclp-cdcl*$_W$*-merge-cp-no-step-cdcl*$_W$*-bj*:
assumes
$conflicting\ S = None$ **and**
$cdcl_W\text{-}merge\text{-}cp^{**}\ S\ T$
shows $no\text{-}step\ cdcl_W\text{-}bj\ T$
⟨*proof*⟩

lemma *conflicting-true-full-cdcl*$_W$*-merge-cp-iff-full-cdcl*$_W$*-s'-without-decode*:
assumes
confl: $conflicting\ S = None$ **and**
inv: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows
$full\ cdcl_W\text{-}merge\text{-}cp\ S\ V \longleftrightarrow full\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S\ V$ (**is** *?fw* $\longleftrightarrow$ *?s'*)

⟨*proof*⟩

lemma *conflicting-true-full1-*$cdcl_W$*-merge-cp-iff-full1-*$cdcl_W$*-s'-without-decode*:
assumes
confl: *conflicting S = None* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows
full1 $cdcl_W$*-merge-cp S V* ⟷ *full1* $cdcl_W$*-s'-without-decide S V*
⟨*proof*⟩

lemma *conflicting-true-full1-*$cdcl_W$*-merge-cp-imp-full1-*$cdcl_W$*-s'-without-decode*:
assumes
fw: *full1* $cdcl_W$*-merge-cp S V* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows
full1 $cdcl_W$*-s'-without-decide S V*
⟨*proof*⟩

inductive $cdcl_W$*-merge-stgy* **for** *S* :: *'st* **where**
fw-s-cp[*intro*]: *full1* $cdcl_W$*-merge-cp S T* ⟹ $cdcl_W$*-merge-stgy S T* |
fw-s-decide[*intro*]: *decide S T* ⟹ *no-step* $cdcl_W$*-merge-cp S* ⟹ *full* $cdcl_W$*-merge-cp T U*
⟹ $cdcl_W$*-merge-stgy S U*

lemma $cdcl_W$*-merge-stgy-tranclp-*$cdcl_W$*-merge*:
assumes *fw*: $cdcl_W$*-merge-stgy S T*
shows $cdcl_W$*-merge*$^{++}$ *S T*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$*-merge*:
assumes *fw*: $cdcl_W$*-merge-stgy*** *S T*
shows $cdcl_W$*-merge*** *S T*
⟨*proof*⟩

lemma $cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge-stgy S T* ⟹ $cdcl_W{}^{**}$ *S T*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge-stgy*** *S T* ⟹ $cdcl_W{}^{**}$ *S T*
⟨*proof*⟩

lemma $cdcl_W$*-merge-stgy-cases*[*consumes 1*, *case-names fw-s-cp fw-s-decide*]:
assumes
$cdcl_W$*-merge-stgy S U*
full1 $cdcl_W$*-merge-cp S U* ⟹ *P*
⋀*T*. *decide S T* ⟹ *no-step* $cdcl_W$*-merge-cp S* ⟹ *full* $cdcl_W$*-merge-cp T U* ⟹ *P*
shows *P*
⟨*proof*⟩

inductive $cdcl_W$*-s'-w* :: *'st* ⇒ *'st* ⇒ *bool* **where**
conflict': *full1* $cdcl_W$*-s'-without-decide S S'* ⟹ $cdcl_W$*-s'-w S S'* |
decide': *decide S S'* ⟹ *no-step* $cdcl_W$*-s'-without-decide S* ⟹ *full* $cdcl_W$*-s'-without-decide S' S''*
⟹ $cdcl_W$*-s'-w S S''*

lemma $cdcl_W$*-s'-w-rtranclp-*$cdcl_W$:
$cdcl_W$*-s'-w S T* ⟹ $cdcl_W{}^{**}$ *S T*

$\langle proof \rangle$

lemma *rtranclp-cdcl$_W$-s'-w-rtranclp-cdcl$_W$*:
$cdcl_W\text{-}s'\text{-}w^{**}\ S\ T \Longrightarrow cdcl_W{}^{**}\ S\ T$
$\langle proof \rangle$

lemma *no-step-cdcl$_W$-cp-no-step-cdcl$_W$-s'-without-decide*:
assumes *no-step cdcl$_W$-cp S* **and** *conflicting S = None* **and** *inv*: *cdcl$_W$-M-level-inv S*
shows *no-step cdcl$_W$-s'-without-decide S*
$\langle proof \rangle$

lemma *no-step-cdcl$_W$-cp-no-step-cdcl$_W$-merge-restart*:
assumes *no-step cdcl$_W$-cp S* **and** *conflicting S = None*
shows *no-step cdcl$_W$-merge-cp S*
$\langle proof \rangle$
lemma *after-cdcl$_W$-s'-without-decide-no-step-cdcl$_W$-cp*:
assumes *cdcl$_W$-s'-without-decide S T*
shows *no-step cdcl$_W$-cp T*
$\langle proof \rangle$

lemma *no-step-cdcl$_W$-s'-without-decide-no-step-cdcl$_W$-cp*:
cdcl$_W$-all-struct-inv S $\Longrightarrow$ *no-step cdcl$_W$-s'-without-decide S* $\Longrightarrow$ *no-step cdcl$_W$-cp S*
$\langle proof \rangle$

lemma *after-cdcl$_W$-s'-w-no-step-cdcl$_W$-cp*:
assumes *cdcl$_W$-s'-w S T* **and** *cdcl$_W$-all-struct-inv S*
shows *no-step cdcl$_W$-cp T*
$\langle proof \rangle$

lemma *rtranclp-cdcl$_W$-s'-w-no-step-cdcl$_W$-cp-or-eq*:
assumes $cdcl_W\text{-}s'\text{-}w^{**}\ S\ T$ **and** *cdcl$_W$-all-struct-inv S*
shows $S = T \lor$ *no-step cdcl$_W$-cp T*
$\langle proof \rangle$

lemma *rtranclp-cdcl$_W$-merge-stgy'-no-step-cdcl$_W$-cp-or-eq*:
assumes $cdcl_W\text{-}merge\text{-}stgy^{**}\ S\ T$ **and** *inv*: *cdcl$_W$-all-struct-inv S*
shows $S = T \lor$ *no-step cdcl$_W$-cp T*
$\langle proof \rangle$

lemma *no-step-cdcl$_W$-s'-without-decide-no-step-cdcl$_W$-bj*:
assumes *no-step cdcl$_W$-s'-without-decide S* **and** *inv*: *cdcl$_W$-all-struct-inv S*
shows *no-step cdcl$_W$-bj S*
$\langle proof \rangle$

lemma *cdcl$_W$-s'-w-no-step-cdcl$_W$-bj*:
assumes *cdcl$_W$-s'-w S T* **and** *cdcl$_W$-all-struct-inv S*
shows *no-step cdcl$_W$-bj T*
$\langle proof \rangle$

lemma *rtranclp-cdcl$_W$-s'-w-no-step-cdcl$_W$-bj-or-eq*:
assumes $cdcl_W\text{-}s'\text{-}w^{**}\ S\ T$ **and** *cdcl$_W$-all-struct-inv S*
shows $S = T \lor$ *no-step cdcl$_W$-bj T*
$\langle proof \rangle$

lemma *rtranclp-cdcl$_W$-s'-no-step-cdcl$_W$-s'-without-decide-decomp-into-cdcl$_W$-merge*:
assumes

$cdcl_W\text{-}s'^{**}$ *R V* **and**
conflicting R = None **and**
inv: $cdcl_W$*-all-struct-inv R*
shows ($cdcl_W$*-merge-stgy*** *R V* ∧ *conflicting V = None*)
∨ ($cdcl_W$*-merge-stgy*** *R V* ∧ *conflicting V* ≠ *None* ∧ *no-step* $cdcl_W$*-bj V*)
∨ (∃ *S T U*. $cdcl_W$*-merge-stgy*** *R S* ∧ *no-step* $cdcl_W$*-merge-cp S* ∧ *decide S T*
∧ $cdcl_W$*-merge-cp*** *T U* ∧ *conflict U V*)
∨ (∃ *S T*. $cdcl_W$*-merge-stgy*** *R S* ∧ *no-step* $cdcl_W$*-merge-cp S* ∧ *decide S T*
∧ $cdcl_W$*-merge-cp*** *T V*
∧ *conflicting V = None*)
∨ ($cdcl_W$*-merge-cp*** *R V* ∧ *conflicting V = None*)
∨ (∃ *U*. $cdcl_W$*-merge-cp*** *R U* ∧ *conflict U V*)
⟨*proof*⟩

lemma *decide-rtranclp-*$cdcl_W\text{-}s'$*-rtranclp-*$cdcl_W\text{-}s'$:
assumes
dec: *decide S T* **and**
$cdcl_W\text{-}s'^{**}$ *T U* **and**
n-s-S: *no-step* $cdcl_W$*-cp S* **and**
no-step $cdcl_W$*-cp U*
shows $cdcl_W\text{-}s'^{**}$ *S U*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W\text{-}s'$:
assumes
$cdcl_W$*-merge-stgy*** *R V* **and**
inv: $cdcl_W$*-all-struct-inv R*
shows $cdcl_W\text{-}s'^{**}$ *R V*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-stgy-distinct-mset-clauses*:
assumes *invR*: $cdcl_W$*-all-struct-inv R* **and**
st: $cdcl_W$*-merge-stgy*** *R S* **and**
dist: *distinct-mset* (*clauses R*) **and**
R: *trail R* = []
shows *distinct-mset* (*clauses S*)
⟨*proof*⟩

lemma *no-step-*$cdcl_W\text{-}s'$*-no-step-*$cdcl_W$*-merge-stgy*:
assumes
inv: $cdcl_W$*-all-struct-inv R* **and** *s'*: *no-step* $cdcl_W\text{-}s'$ *R*
shows *no-step* $cdcl_W$*-merge-stgy R*
⟨*proof*⟩
end

Termination and full Equivalence

We will discharge the assumption later using NOT's proof of termination.

locale *conflict-driven-clause-learning*$_W$*-termination* =
conflict-driven-clause-learning$_W$ +
assumes *wf-*$cdcl_W$*-merge-inv*: *wf* {(*T*, *S*). $cdcl_W$*-all-struct-inv S* ∧ $cdcl_W$*-merge S T*}
begin

lemma *wf-tranclp-*$cdcl_W$*-merge*: *wf* {(*T*, *S*). $cdcl_W$*-all-struct-inv S* ∧ $cdcl_W$*-merge*$^{++}$ *S T*}
⟨*proof*⟩

lemma *wf-cdcl$_W$-merge-cp*:
  wf{(*T*, *S*). *cdcl$_W$-all-struct-inv S* ∧ *cdcl$_W$-merge-cp S T*}
  ⟨*proof*⟩

lemma *wf-cdcl$_W$-merge-stgy*:
  wf{(*T*, *S*). *cdcl$_W$-all-struct-inv S* ∧ *cdcl$_W$-merge-stgy S T*}
  ⟨*proof*⟩

lemma *cdcl$_W$-merge-cp-obtain-normal-form*:
  assumes *inv*: *cdcl$_W$-all-struct-inv R*
  obtains *S* **where** *full cdcl$_W$-merge-cp R S*
⟨*proof*⟩

lemma *no-step-cdcl$_W$-merge-stgy-no-step-cdcl$_W$-s'*:
  assumes
    inv: *cdcl$_W$-all-struct-inv R* **and**
    confl: *conflicting R* = *None* **and**
    n-s: *no-step cdcl$_W$-merge-stgy R*
  shows *no-step cdcl$_W$-s' R*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-cp-no-step-cdcl$_W$-bj*:
  assumes *conflicting R* = *None* **and** *cdcl$_W$-merge-cp** R S*
  shows *no-step cdcl$_W$-bj S*
  ⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-stgy-no-step-cdcl$_W$-bj*:
  assumes *confl*: *conflicting R* = *None* **and** *cdcl$_W$-merge-stgy** R S*
  shows *no-step cdcl$_W$-bj S*
  ⟨*proof*⟩

end

end
theory *CDCL-WNOT*
imports *CDCL-NOT CDCL-W-Termination CDCL-W-Merge*
begin

6.3 Link between Weidenbach's and NOT's CDCL

6.3.1 Inclusion of the states

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

fun *convert-ann-lit-from-W* **where**
convert-ann-lit-from-W (*Propagated L* -) = *Propagated L* () |
convert-ann-lit-from-W (*Decided L*) = *Decided L*

abbreviation *convert-trail-from-W* ::
  (*'v*, *'mark*) *ann-lits*
    ⇒ (*'v*, *unit*) *ann-lits* **where**
convert-trail-from-W ≡ *map convert-ann-lit-from-W*

lemma *lits-of-l-convert-trail-from-W*[*simp*]:

lits-of-l (*convert-trail-from-W M*) = *lits-of-l M*
⟨*proof*⟩

lemma *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-ann-lit-from-W L*) = *lit-of L*
⟨*proof*⟩

lemma *no-dup-convert-from-W*[*simp*]:
no-dup (*convert-trail-from-W M*) ⟷ *no-dup M*
⟨*proof*⟩

lemma *convert-trail-from-W-true-annots*[*simp*]:
convert-trail-from-W M ⊨*as C* ⟷ *M* ⊨*as C*
⟨*proof*⟩

lemma *defined-lit-convert-trail-from-W*[*simp*]:
defined-lit (*convert-trail-from-W S*) *L* ⟷ *defined-lit S L*
⟨*proof*⟩

The values *0* and {#} are dummy values.

consts *dummy-cls* :: *'cls*
fun *convert-ann-lit-from-NOT*
:: (*'v*, *'mark*) *ann-lit* ⇒ (*'v*, *'cls*) *ann-lit* **where**
convert-ann-lit-from-NOT (*Propagated L -*) = *Propagated L dummy-cls* |
convert-ann-lit-from-NOT (*Decided L*) = *Decided L*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT ≡ *map convert-ann-lit-from-NOT*

lemma *undefined-lit-convert-trail-from-NOT*[*simp*]:
undefined-lit (*convert-trail-from-NOT F*) *L* ⟷ *undefined-lit F L*
⟨*proof*⟩

lemma *lits-of-l-convert-trail-from-NOT*:
lits-of-l (*convert-trail-from-NOT F*) = *lits-of-l F*
⟨*proof*⟩

lemma *convert-trail-from-W-from-NOT*[*simp*]:
convert-trail-from-W (*convert-trail-from-NOT M*) = *M*
⟨*proof*⟩

lemma *convert-trail-from-W-convert-lit-from-NOT*[*simp*]:
convert-ann-lit-from-W (*convert-ann-lit-from-NOT L*) = *L*
⟨*proof*⟩

abbreviation $trail_{NOT}$ **where**
$trail_{NOT}$ *S* ≡ *convert-trail-from-W* (*fst S*)

lemma *undefined-lit-convert-trail-from-W*[*iff*]:
undefined-lit (*convert-trail-from-W M*) *L* ⟷ *undefined-lit M L*
⟨*proof*⟩

lemma *lit-of-convert-ann-lit-from-NOT*[*iff*]:
lit-of (*convert-ann-lit-from-NOT L*) = *lit-of L*
⟨*proof*⟩

sublocale $state_W \subseteq$ *dpll-state-ops*
$\lambda S.$ *convert-trail-from-W* (*trail* S)
clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
⟨*proof*⟩

sublocale $state_W \subseteq$ *dpll-state*
$\lambda S.$ *convert-trail-from-W* (*trail* S)
clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
⟨*proof*⟩

context $state_W$
begin
declare $state\text{-}simp_{NOT}$[*simp del*]
end

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}$*-merge-bj-learn-ops*
$\lambda S.$ *convert-trail-from-W* (*trail* S)
clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
λ- -. *True*
λ- $S.$ *conflicting* S = *None*
$\lambda C\ C'\ L'\ S\ T.$ *backjump-l-cond* $C\ C'\ L'\ S\ T$
$\wedge$ *distinct-mset* $(C' + \{\#L'\#\}) \wedge \neg$*tautology* $(C' + \{\#L'\#\})$
⟨*proof*⟩

thm $cdcl_{NOT}$*-merge-bj-learn-proxy.axioms*
sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}$*-merge-bj-learn-proxy*
$\lambda S.$ *convert-trail-from-W* (*trail* S)
clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$

λ- -. *True*
λ- $S.$ *conflicting* S = *None*
backjump-l-cond
inv_{NOT}
⟨*proof*⟩

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}$*-merge-bj-learn-proxy2*
$\lambda S.$ *convert-trail-from-W* (*trail* S)
clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S

$\lambda C\ S.\ add\text{-}learned\text{-}cls\ C\ S$
$\lambda C\ S.\ remove\text{-}cls\ C\ S$
$\lambda\text{-}\ \text{-}.\ True$
$\lambda\text{-}\ S.\ conflicting\ S = None\ backjump\text{-}l\text{-}cond\ inv_{NOT}$
$\langle proof \rangle$

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn$
$\lambda S.\ convert\text{-}trail\text{-}from\text{-}W\ (trail\ S)$
$clauses$
$\lambda L\ S.\ cons\text{-}trail\ (convert\text{-}ann\text{-}lit\text{-}from\text{-}NOT\ L)\ S$
$\lambda S.\ tl\text{-}trail\ S$
$\lambda C\ S.\ add\text{-}learned\text{-}cls\ C\ S$
$\lambda C\ S.\ remove\text{-}cls\ C\ S$
$backjump\text{-}l\text{-}cond$
$\lambda\text{-}\ \text{-}.\ True$
$\lambda\text{-}\ S.\ conflicting\ S = None\ inv_{NOT}$
$\langle proof \rangle$

context $conflict\text{-}driven\text{-}clause\text{-}learning_W$
begin

Notations are lost while proving locale inclusion:

notation $state\text{-}eq_{NOT}$ (**infix** $\sim_{NOT}$ 50)

6.3.2 Additional Lemmas between NOT and W states

lemma $trail_W\text{-}eq\text{-}reduce\text{-}trail\text{-}to_{NOT}\text{-}eq$:
$trail\ S = trail\ T \Longrightarrow trail\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ S) = trail\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ T)$
$\langle proof \rangle$

lemma $trail\text{-}reduce\text{-}trail\text{-}to_{NOT}\text{-}add\text{-}learned\text{-}cls$:
$no\text{-}dup\ (trail\ S) \Longrightarrow$
$trail\ (reduce\text{-}trail\text{-}to_{NOT}\ M\ (add\text{-}learned\text{-}cls\ D\ S)) = trail\ (reduce\text{-}trail\text{-}to_{NOT}\ M\ S)$
$\langle proof \rangle$

lemma $reduce\text{-}trail\text{-}to_{NOT}\text{-}reduce\text{-}trail\text{-}convert$:
$reduce\text{-}trail\text{-}to_{NOT}\ C\ S = reduce\text{-}trail\text{-}to\ (convert\text{-}trail\text{-}from\text{-}NOT\ C)\ S$
$\langle proof \rangle$

lemma $reduce\text{-}trail\text{-}to\text{-}map[simp]$:
$reduce\text{-}trail\text{-}to\ (map\ f\ M)\ S = reduce\text{-}trail\text{-}to\ M\ S$
$\langle proof \rangle$

lemma $reduce\text{-}trail\text{-}to_{NOT}\text{-}map[simp]$:
$reduce\text{-}trail\text{-}to_{NOT}\ (map\ f\ M)\ S = reduce\text{-}trail\text{-}to_{NOT}\ M\ S$
$\langle proof \rangle$

lemma $skip\text{-}or\text{-}resolve\text{-}state\text{-}change$:
assumes $skip\text{-}or\text{-}resolve^{**}\ S\ T$
shows
$\exists M.\ trail\ S = M\ @\ trail\ T \wedge (\forall m \in set\ M.\ \neg is\text{-}decided\ m)$
$clauses\ S = clauses\ T$
$backtrack\text{-}lvl\ S = backtrack\text{-}lvl\ T$
$\langle proof \rangle$

6.3.3 Inclusion of Weidenbach's CDCL in NOT's CDCL

This lemma shows the inclusion of Weidenbach's CDCL $cdcl_W$*-merge* (with merging) in NOT's $cdcl_{NOT}$*-merged-bj-learn*.

lemma $cdcl_W$*-merge-is-*$cdcl_{NOT}$*-merged-bj-learn*:
  assumes
    inv: $cdcl_W$*-all-struct-inv* S **and**
    $cdcl_W$: $cdcl_W$*-merge* S T
  shows $cdcl_{NOT}$*-merged-bj-learn* S T
    $\vee$ (*no-step* $cdcl_W$*-merge* T $\wedge$ *conflicting* $T \neq$ *None*)
  ⟨*proof*⟩

abbreviation $cdcl_{NOT}$*-restart* **where**
$cdcl_{NOT}$*-restart* $\equiv$ *restart-ops.*$cdcl_{NOT}$*-raw-restart* $cdcl_{NOT}$ *restart*

lemma $cdcl_W$*-merge-restart-is-*$cdcl_{NOT}$*-merged-bj-learn-restart-no-step*:
  assumes
    inv: $cdcl_W$*-all-struct-inv* S **and**
    $cdcl_W$:$cdcl_W$*-merge-restart* S T
  shows $cdcl_{NOT}$*-restart*** S T $\vee$ (*no-step* $cdcl_W$*-merge* T $\wedge$ *conflicting* $T \neq$ *None*)
⟨*proof*⟩

abbreviation μ_{FW} :: *'st* $\Rightarrow$ *nat* **where**
μ_{FW} S $\equiv$ (*if no-step* $cdcl_W$*-merge* S *then 0 else 1+*μ_{CDCL}*'-merged* (*set-mset* (*init-clss* S)) S)

lemma $cdcl_W$*-merge-*μ_{FW}*-decreasing*:
  assumes
    inv: $cdcl_W$*-all-struct-inv* S **and**
    fw: $cdcl_W$*-merge* S T
  shows μ_{FW} T $<$ μ_{FW} S
⟨*proof*⟩

lemma *wf-*$cdcl_W$*-merge*: *wf* $\{(T, S)$. $cdcl_W$*-all-struct-inv* S $\wedge$ $cdcl_W$*-merge* S $T\}$
  ⟨*proof*⟩

sublocale *conflict-driven-clause-learning*$_W$*-termination*
  ⟨*proof*⟩

6.3.4 Correctness of $cdcl_W$*-merge-stgy*

lemma *full-*$cdcl_W$*-s'-full-*$cdcl_W$*-merge-restart*:
  assumes
    conflicting R = *None* **and**
    inv: $cdcl_W$*-all-struct-inv* R
  shows *full* $cdcl_W$*-s'* R V $\longleftrightarrow$ *full* $cdcl_W$*-merge-stgy* R V (**is** *?s'* $\longleftrightarrow$ *?fw*)
⟨*proof*⟩

lemma *full-*$cdcl_W$*-stgy-full-*$cdcl_W$*-merge*:
  assumes
    conflicting R = *None* **and**
    $cdcl_W$*-all-struct-inv* R
  shows *full* $cdcl_W$*-stgy* R V $\longleftrightarrow$ *full* $cdcl_W$*-merge-stgy* R V
  ⟨*proof*⟩

lemma *full-*$cdcl_W$*-merge-stgy-final-state-conclusive'*:
  fixes S' :: *'st*

assumes
full: *full* $cdcl_W$*-merge-stgy* (*init-state N*) S' **and**
no-d: *distinct-mset-mset N*
shows (*conflicting* S' = *Some* {#} ∧ *unsatisfiable* (*set-mset N*))
∨ (*conflicting* S' = *None* ∧ *trail* S' ⊨*asm N* ∧ *satisfiable* (*set-mset N*))
⟨*proof*⟩
end

end
theory *CDCL-W-Restart*
imports *CDCL-W-Merge*
begin

6.3.5 Adding Restarts

locale $cdcl_W$*-restart* =
conflict-driven-clause-learning$_W$
— functions for the state:
— access functions:
trail init-clss learned-clss backtrack-lvl conflicting
— changing state:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state
for
trail :: '*st* ⇒ ('*v*, '*v clause*) *ann-lits* **and**
init-clss :: '*st* ⇒ '*v clauses* **and**
learned-clss :: '*st* ⇒ '*v clauses* **and**
backtrack-lvl :: '*st* ⇒ *nat* **and**
conflicting :: '*st* ⇒ '*v clause option* **and**

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

init-state :: '*v clauses* ⇒ '*st* +
fixes *f* :: *nat* ⇒ *nat*
assumes *f*: *unbounded f*
begin

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive $cdcl_W$*-merge-with-restart* **where**
restart-step:
($cdcl_W$*-stgy*^^(*card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)))) *S T*
⟹ *card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)) > *f n*
⟹ *restart T U* ⟹ $cdcl_W$*-merge-with-restart* (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1* $cdcl_W$*-stgy S T* ⟹ $cdcl_W$*-merge-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma $cdcl_W$*-merge-with-restart-rtranclp-cdcl*$_W$:

$cdcl_W$-merge-with-restart S $T \Longrightarrow cdcl_W^{**}$ (fst S) (fst T)
⟨proof⟩

lemma $cdcl_W$-merge-with-restart-increasing-number:
$cdcl_W$-merge-with-restart S $T \Longrightarrow$ snd T = 1 + snd S
⟨proof⟩

lemma full1 $cdcl_W$-stgy S $T \Longrightarrow cdcl_W$-merge-with-restart (S, n) (T, Suc n)
⟨proof⟩

lemma $cdcl_W$-all-struct-inv-learned-clss-bound:
assumes inv: $cdcl_W$-all-struct-inv S
shows set-mset (learned-clss S) $\subseteq$ simple-clss (atms-of-mm (init-clss S))
⟨proof⟩

lemma $cdcl_W$-merge-with-restart-init-clss:
$cdcl_W$-merge-with-restart S $T \Longrightarrow cdcl_W$-M-level-inv (fst S) $\Longrightarrow$
init-clss (fst S) = init-clss (fst T)
⟨proof⟩

lemma
wf $\{(T, S)$. $cdcl_W$-all-struct-inv (fst S) $\wedge$ $cdcl_W$-merge-with-restart S $T\}$
⟨proof⟩

lemma $cdcl_W$-merge-with-restart-distinct-mset-clauses:
assumes invR: $cdcl_W$-all-struct-inv (fst R) **and**
st: $cdcl_W$-merge-with-restart R S **and**
dist: distinct-mset (clauses (fst R)) **and**
R: trail (fst R) = []
shows distinct-mset (clauses (fst S))
⟨proof⟩

inductive $cdcl_W$-with-restart **where**
restart-step:
($cdcl_W$-stgy^^(card (set-mset (learned-clss T)) $-$ card (set-mset (learned-clss S)))) S $T \Longrightarrow$
card (set-mset (learned-clss T)) $-$ card (set-mset (learned-clss S)) $>$ f $n \Longrightarrow$
restart T $U \Longrightarrow$
$cdcl_W$-with-restart (S, n) (U, Suc n) |
restart-full: full1 $cdcl_W$-stgy S $T \Longrightarrow cdcl_W$-with-restart (S, n) (T, Suc n)

lemma $cdcl_W$-with-restart-rtranclp-$cdcl_W$:
$cdcl_W$-with-restart S $T \Longrightarrow cdcl_W^{**}$ (fst S) (fst T)
⟨proof⟩

lemma $cdcl_W$-with-restart-increasing-number:
$cdcl_W$-with-restart S $T \Longrightarrow$ snd T = 1 + snd S
⟨proof⟩

lemma full1 $cdcl_W$-stgy S $T \Longrightarrow cdcl_W$-with-restart (S, n) (T, Suc n)
⟨proof⟩

lemma $cdcl_W$-with-restart-init-clss:
$cdcl_W$-with-restart S $T \Longrightarrow cdcl_W$-M-level-inv (fst S) $\Longrightarrow$ init-clss (fst S) = init-clss (fst T)
⟨proof⟩

lemma

$wf\ \{(T, S).\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ (fst\ S) \wedge cdcl_W\text{-}with\text{-}restart\ S\ T\}$
⟨proof⟩

lemma *cdcl*$_W$*-with-restart-distinct-mset-clauses*:
assumes *invR*: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ (fst\ R)$ **and**
st: $cdcl_W\text{-}with\text{-}restart\ R\ S$ **and**
dist: $distinct\text{-}mset\ (clauses\ (fst\ R))$ **and**
R: $trail\ (fst\ R) = []$
shows $distinct\text{-}mset\ (clauses\ (fst\ S))$
⟨proof⟩
end

locale *luby-sequence* =
fixes $ur :: nat$
assumes $ur > 0$
begin

lemma *exists-luby-decomp*:
fixes $i :: nat$
shows $\exists k::nat.\ (2\ \hat{}\ (k - 1) \leq i \wedge i < 2\ \hat{}\ k - 1) \vee i = 2\ \hat{}\ k - 1$
⟨proof⟩

Luby sequences are defined by:

- $2^k - 1$, if $i = (2::'a)^k - (1::'a)$
- $luby\text{-}sequence\text{-}core\ (i - 2^{k-1} + 1)$, if $(2::'a)^{k-1} \leq i$ and $i \leq (2::'a)^k - (1::'a)$

Then the sequence is then scaled by a constant unit run (called *ur* here), strictly positive.

function $luby\text{-}sequence\text{-}core :: nat \Rightarrow nat$ **where**
$luby\text{-}sequence\text{-}core\ i =$
$(if\ \exists k.\ i = 2\hat{}k - 1$
$then\ 2\hat{}((SOME\ k.\ i = 2\hat{}k - 1) - 1)$
$else\ luby\text{-}sequence\text{-}core\ (i - 2\hat{}((SOME\ k.\ 2\hat{}(k-1) \leq i \wedge i < 2\hat{}k - 1) - 1) + 1))$
⟨proof⟩
termination
⟨proof⟩

function $natlog2 :: nat \Rightarrow nat$ **where**
$natlog2\ n = (if\ n = 0\ then\ 0\ else\ 1 + natlog2\ (n\ div\ 2))$
⟨proof⟩
termination ⟨proof⟩

declare *natlog2.simps*[*simp del*]

declare *luby-sequence-core.simps*[*simp del*]

lemma *two-pover-n-eq-two-power-n'-eq*:
assumes *H*: $(2::nat)\ \hat{}\ (k::nat) - 1 = 2\ \hat{}\ k' - 1$
shows $k' = k$
⟨proof⟩

lemma *luby-sequence-core-two-power-minus-one*:
$luby\text{-}sequence\text{-}core\ (2\hat{}k - 1) = 2\hat{}(k-1)$ (**is** $?L = ?K$)
⟨proof⟩

lemma *different-luby-decomposition-false*:
  assumes
    H: $2 \,\hat{}\, (k - Suc\ 0) \leq i$ **and**
    k': $i < 2 \,\hat{}\, k' - Suc\ 0$ **and**
    k-k': $k > k'$
  shows *False*
⟨*proof*⟩

lemma *luby-sequence-core-not-two-power-minus-one*:
  assumes
    k-i: $2 \,\hat{}\, (k - 1) \leq i$ **and**
    i-k: $i < 2\hat{}\, k - 1$
  shows *luby-sequence-core* i = *luby-sequence-core* $(i - 2 \,\hat{}\, (k - 1) + 1)$
⟨*proof*⟩

lemma *unbounded-luby-sequence-core*: *unbounded luby-sequence-core*
  ⟨*proof*⟩

abbreviation *luby-sequence* :: *nat* ⇒ *nat* **where**
luby-sequence n ≡ *ur* ∗ *luby-sequence-core n*

lemma *bounded-luby-sequence*: *unbounded luby-sequence*
  ⟨*proof*⟩

lemma *luby-sequence-core-0*: *luby-sequence-core* $0 = 1$
⟨*proof*⟩

lemma *luby-sequence-core n* ≥ 1
⟨*proof*⟩
end

locale *luby-sequence-restart* =
  luby-sequence ur +
  conflict-driven-clause-learning$_W$
    — functions for the state:
      — access functions:
    trail init-clss learned-clss backtrack-lvl conflicting
      — changing state:
    cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
    update-conflicting

      — get state:
    init-state
  for
    ur :: *nat* **and**
    trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
    hd-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lit* **and**
    init-clss :: *'st* ⇒ *'v clauses* **and**
    learned-clss :: *'st* ⇒ *'v clauses* **and**
    backtrack-lvl :: *'st* ⇒ *nat* **and**
    conflicting :: *'st* ⇒ *'v clause option* **and**

    cons-trail :: (*'v*, *'v clause*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
    tl-trail :: *'st* ⇒ *'st* **and**
    add-learned-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**

remove-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'v clause option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'v clauses* ⇒ *'st*
begin

sublocale $cdcl_W$*-restart* - - - - - - - - - - - - *luby-sequence*
⟨*proof*⟩

end
end
theory *CDCL-W-Incremental*
imports *CDCL-W-Termination*
begin

6.4 Incremental SAT solving

locale $state_W$*-adding-init-clause* =
$state_W$
— functions about the state:
— getter:
trail init-clss learned-clss backtrack-lvl conflicting
— setter:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— Some specific states:
init-state
for
trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
init-clss :: *'st* ⇒ *'v clauses* **and**
learned-clss :: *'st* ⇒ *'v clauses* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
conflicting :: *'st* ⇒ *'v clause option* **and**

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

init-state :: *'v clauses* ⇒ *'st* +
fixes
add-init-cls :: *'v clause* ⇒ *'st* ⇒ *'st*
assumes
add-init-cls:
state st = (M, N, U, S') ⟹
state (*add-init-cls C st*) = $(M, \{\#C\#\} + N, U, S')$
begin

lemma
trail-add-init-cls[*simp*]:
trail (*add-init-cls C st*) = *trail st* **and**

init-clss-add-init-cls[*simp*]:
init-clss (*add-init-cls* *C* *st*) = {#*C*#} + *init-clss* *st*
and
learned-clss-add-init-cls[*simp*]:
learned-clss (*add-init-cls* *C* *st*) = *learned-clss* *st* **and**
backtrack-lvl-add-init-cls[*simp*]:
backtrack-lvl (*add-init-cls* *C* *st*) = *backtrack-lvl* *st* **and**
conflicting-add-init-cls[*simp*]:
conflicting (*add-init-cls* *C* *st*) = *conflicting* *st*
⟨*proof*⟩

lemma *clauses-add-init-cls*[*simp*]:
clauses (*add-init-cls* *N* *S*) = {#*N*#} + *init-clss* *S* + *learned-clss* *S*
⟨*proof*⟩

lemma *reduce-trail-to-add-init-cls*[*simp*]:
trail (*reduce-trail-to* *F* (*add-init-cls* *C* *S*)) = *trail* (*reduce-trail-to* *F* *S*)
⟨*proof*⟩

lemma *conflicting-add-init-cls-iff-conflicting*[*simp*]:
conflicting (*add-init-cls* *C* *S*) = *None* ⟷ *conflicting* *S* = *None*
⟨*proof*⟩
end

locale *conflict-driven-clause-learning-with-adding-init-clause*$_W$ =
state$_W$*-adding-init-clause*

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

— get state:
init-state
— Adding a clause:
add-init-cls
for
trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lit* **and**
init-clss :: *'st* ⇒ *'v clauses* **and**
learned-clss :: *'st* ⇒ *'v clauses* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
conflicting :: *'st* ⇒ *'v clause option* **and**

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

init-state :: *'v clauses* ⇒ *'st* **and**
add-init-cls :: *'v clause* ⇒ *'st* ⇒ *'st*
begin

sublocale *conflict-driven-clause-learning*$_W$
⟨*proof*⟩

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl*$_W$*-all-struct-inv*

definition *cdcl*$_W$*-stgy-invariant* **where**
cdcl$_W$*-stgy-invariant S* ⟷
conflict-is-false-with-level S
∧ *no-clause-is-false S*
∧ *no-smaller-confl S*
∧ *no-clause-is-false S*

lemma *cdcl*$_W$*-stgy-cdcl*$_W$*-stgy-invariant*:
assumes
cdcl$_W$: *cdcl*$_W$*-stgy S T* **and**
inv-s: *cdcl*$_W$*-stgy-invariant S* **and**
inv: *cdcl*$_W$*-all-struct-inv S*
shows
cdcl$_W$*-stgy-invariant T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-cdcl*$_W$*-stgy-invariant*:
assumes
cdcl$_W$: *cdcl*$_W$*-stgy*** *S T* **and**
inv-s: *cdcl*$_W$*-stgy-invariant S* **and**
inv: *cdcl*$_W$*-all-struct-inv S*
shows
cdcl$_W$*-stgy-invariant T*
⟨*proof*⟩

abbreviation *decr-bt-lvl* **where**
decr-bt-lvl S ≡ *update-backtrack-lvl* (*backtrack-lvl S* − *1*) *S*

When we add a new clause, we reduce the trail until we get to tho first literal included in C. Then we can mark the conflict.

fun *cut-trail-wrt-clause* **where**
cut-trail-wrt-clause C [] *S* = *S* |
cut-trail-wrt-clause C (*Decided L* # *M*) *S* =
(*if* −*L* ∈# *C then S*
else cut-trail-wrt-clause C M (*decr-bt-lvl* (*tl-trail S*))) |
cut-trail-wrt-clause C (*Propagated L* - # *M*) *S* =
(*if* −*L* ∈# *C then S*
else cut-trail-wrt-clause C M (*tl-trail S*))

definition *add-new-clause-and-update* :: *'v clause* ⇒ *'st* ⇒ *'st* **where**
add-new-clause-and-update C S =
(*if trail S* ⊨*as CNot C*
then update-conflicting (*Some C*) (*add-init-cls C*
(*cut-trail-wrt-clause C* (*trail S*) *S*))
else add-init-cls C S)

thm *cut-trail-wrt-clause.induct*
lemma *init-clss-cut-trail-wrt-clause*[*simp*]:
init-clss (*cut-trail-wrt-clause C M S*) = *init-clss S*

⟨*proof*⟩

lemma *learned-clss-cut-trail-wrt-clause*[*simp*]:
learned-clss (*cut-trail-wrt-clause* C M S) = *learned-clss* S
⟨*proof*⟩

lemma *conflicting-clss-cut-trail-wrt-clause*[*simp*]:
conflicting (*cut-trail-wrt-clause* C M S) = *conflicting* S
⟨*proof*⟩

lemma *trail-cut-trail-wrt-clause*:
$\exists M$. *trail* S = M @ *trail* (*cut-trail-wrt-clause* C (*trail* S) S)
⟨*proof*⟩

lemma *n-dup-no-dup-trail-cut-trail-wrt-clause*[*simp*]:
assumes *n-d*: *no-dup* (*trail* T)
shows *no-dup* (*trail* (*cut-trail-wrt-clause* C (*trail* T) T))
⟨*proof*⟩

lemma *cut-trail-wrt-clause-backtrack-lvl-length-decided*:
assumes
backtrack-lvl T = *count-decided* (*trail* T)
shows
backtrack-lvl (*cut-trail-wrt-clause* C (*trail* T) T) =
count-decided (*trail* (*cut-trail-wrt-clause* C (*trail* T) T))
⟨*proof*⟩

lemma *cut-trail-wrt-clause-CNot-trail*:
assumes *trail* T $\models$*as* *CNot* C
shows
(*trail* ((*cut-trail-wrt-clause* C (*trail* T) T))) $\models$*as* *CNot* C
⟨*proof*⟩

lemma *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*:
(($\forall L \in\# C$. $-L \notin$ *lits-of-l* (*trail* T)) $\wedge$ *trail* (*cut-trail-wrt-clause* C (*trail* T) T) = [])
$\vee$ ($-$*lit-of* (*hd* (*trail* (*cut-trail-wrt-clause* C (*trail* T) T))) $\in\#$ C
$\wedge$ *length* (*trail* (*cut-trail-wrt-clause* C (*trail* T) T)) $\geq$ *1*)
⟨*proof*⟩

We can fully run $cdcl_W$-*s* or add a clause. Remark that we use $cdcl_W$-*s* to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict C if possible.

inductive *incremental-cdcl*$_W$ :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** S **where**
add-confl:
trail S $\models$*asm* *init-clss* S $\Longrightarrow$ *distinct-mset* C $\Longrightarrow$ *conflicting* S = *None* $\Longrightarrow$
trail S $\models$*as* *CNot* C $\Longrightarrow$
full $cdcl_W$-*stgy*
(*update-conflicting* (*Some* C)
(*add-init-cls* C (*cut-trail-wrt-clause* C (*trail* S) S))) T $\Longrightarrow$
incremental-cdcl$_W$ S T |
add-no-confl:
trail S $\models$*asm* *init-clss* S $\Longrightarrow$ *distinct-mset* C $\Longrightarrow$ *conflicting* S = *None* $\Longrightarrow$
$\neg$*trail* S $\models$*as* *CNot* C $\Longrightarrow$
full $cdcl_W$-*stgy* (*add-init-cls* C S) T $\Longrightarrow$
incremental-cdcl$_W$ S T

lemma $cdcl_W$-*all-struct-inv-add-new-clause-and-update-cdcl*$_W$-*all-struct-inv*:

assumes
inv-T: $cdcl_W$*-all-struct-inv T* **and**
tr-T-N[*simp*]: *trail T* $\models$*asm N* **and**
tr-C[*simp*]: *trail T* $\models$*as CNot C* **and**
[*simp*]: *distinct-mset C*
shows $cdcl_W$*-all-struct-inv* (*add-new-clause-and-update C T*) (**is** $cdcl_W$*-all-struct-inv ?T'*)
⟨*proof*⟩

lemma $cdcl_W$*-all-struct-inv-add-new-clause-and-update-*$cdcl_W$*-stgy-inv*:
assumes
inv-s: $cdcl_W$*-stgy-invariant T* **and**
inv: $cdcl_W$*-all-struct-inv T* **and**
tr-T-N[*simp*]: *trail T* $\models$*asm N* **and**
tr-C[*simp*]: *trail T* $\models$*as CNot C* **and**
[*simp*]: *distinct-mset C*
shows $cdcl_W$*-stgy-invariant* (*add-new-clause-and-update C T*)
(**is** $cdcl_W$*-stgy-invariant ?T'*)
⟨*proof*⟩

lemma *full-*$cdcl_W$*-stgy-inv-normal-form*:
assumes
full: *full* $cdcl_W$*-stgy S T* **and**
inv-s: $cdcl_W$*-stgy-invariant S* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows *conflicting T = Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S*))
∨ *conflicting T = None* ∧ *trail T* $\models$*asm init-clss S* ∧ *satisfiable* (*set-mset* (*init-clss S*))
⟨*proof*⟩

lemma *incremental-*$cdcl_W$*-inv*:
assumes
inc: *incremental-*$cdcl_W$ *S T* **and**
inv: $cdcl_W$*-all-struct-inv S* **and**
s-inv: $cdcl_W$*-stgy-invariant S*
shows
$cdcl_W$*-all-struct-inv T* **and**
$cdcl_W$*-stgy-invariant T*
⟨*proof*⟩

lemma *rtranclp-incremental-*$cdcl_W$*-inv*:
assumes
inc: *incremental-*$cdcl_W^{**}$ *S T* **and**
inv: $cdcl_W$*-all-struct-inv S* **and**
s-inv: $cdcl_W$*-stgy-invariant S*
shows
$cdcl_W$*-all-struct-inv T* **and**
$cdcl_W$*-stgy-invariant T*
⟨*proof*⟩

lemma *incremental-conclusive-state*:
assumes
inc: *incremental-*$cdcl_W$ *S T* **and**
inv: $cdcl_W$*-all-struct-inv S* **and**
s-inv: $cdcl_W$*-stgy-invariant S*
shows *conflicting T = Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss T*))
∨ *conflicting T = None* ∧ *trail T* $\models$*asm init-clss T* ∧ *satisfiable* (*set-mset* (*init-clss T*))
⟨*proof*⟩

lemma *tranclp-incremental-correct*:
  assumes
    inc: *incremental-cdcl*$_W$$^{++}$ *S T* **and**
    inv: *cdcl*$_W$*-all-struct-inv S* **and**
    s-inv: *cdcl*$_W$*-stgy-invariant S*
  shows *conflicting T* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss T*))
    ∨ *conflicting T* = *None* ∧ *trail T* ⊨*asm init-clss T* ∧ *satisfiable* (*set-mset* (*init-clss T*))
  ⟨*proof*⟩

end

end
theory *DPLL-CDCL-W-Implementation*
imports *Partial-Annotated-Clausal-Logic CDCL-W-Level*
begin

Chapter 7

Implementation of DPLL and CDCL

We then reuse all the theorems to go towards an implementation using 2-watched literals:

- `CDCL_W_Abstract_State.thy` defines a better-suited state: the operation operating on it are more constrained, allowing simpler proofs and less edge cases later.

7.1 Simple List-Based Implementation of the DPLL and CDCL

The idea of the list-based implementation is to test the stack: the theories about the calculi, adapting the theorems to a simple implementation and the code exportation. The implementation are very simple ans simply iterate over-and-over on lists.

7.1.1 Common Rules

Propagation

The following theorem holds:

lemma *lits-of-l-unfold*[*iff*]:
$(\forall\, c \in \mathit{set}\ C.\ -c \in \mathit{lits\text{-}of\text{-}l}\ \mathit{Ms}) \longleftrightarrow \mathit{Ms} \models\!\mathit{as}\ \mathit{CNot}\ (\mathit{mset}\ C)$
$\langle \mathit{proof} \rangle$

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: $'a\ \mathit{literal}\ \mathit{list} \Rightarrow ('a, 'b)\ \mathit{ann\text{-}lits} \Rightarrow 'a\ \mathit{literal}\ \mathit{option}$
where
$\mathit{is\text{-}unit\text{-}clause}\ l\ M =$
$(\mathit{case}\ \mathit{List.filter}\ (\lambda a.\ \mathit{atm\text{-}of}\ a \notin \mathit{atm\text{-}of}\ `\ \mathit{lits\text{-}of\text{-}l}\ M)\ l\ \mathit{of}$
$a \# [] \Rightarrow \mathit{if}\ M \models\!\mathit{as}\ \mathit{CNot}\ (\mathit{mset}\ l - \{\#a\#\})\ \mathit{then}\ \mathit{Some}\ a\ \mathit{else}\ \mathit{None}$
$|\ \text{-} \Rightarrow \mathit{None})$

definition *is-unit-clause-code* :: $'a\ \mathit{literal}\ \mathit{list} \Rightarrow ('a, 'b)\ \mathit{ann\text{-}lits}$
$\Rightarrow 'a\ \mathit{literal}\ \mathit{option}$ **where**
$\mathit{is\text{-}unit\text{-}clause\text{-}code}\ l\ M =$
$(\mathit{case}\ \mathit{List.filter}\ (\lambda a.\ \mathit{atm\text{-}of}\ a \notin \mathit{atm\text{-}of}\ `\ \mathit{lits\text{-}of\text{-}l}\ M)\ l\ \mathit{of}$
$a \# [] \Rightarrow \mathit{if}\ (\forall\, c \in\!\mathit{set}\ (\mathit{remove1}\ a\ l).\ -c \in \mathit{lits\text{-}of\text{-}l}\ M)\ \mathit{then}\ \mathit{Some}\ a\ \mathit{else}\ \mathit{None}$
$|\ \text{-} \Rightarrow \mathit{None})$

lemma *is-unit-clause-is-unit-clause-code*[*code*]:
$\mathit{is\text{-}unit\text{-}clause}\ l\ M = \mathit{is\text{-}unit\text{-}clause\text{-}code}\ l\ M$

⟨*proof*⟩

lemma *is-unit-clause-some-undef*:
  assumes *is-unit-clause l M = Some a*
  shows *undefined-lit M a*
⟨*proof*⟩

lemma *is-unit-clause-some-CNot*: *is-unit-clause l M = Some a* $\Longrightarrow$ *M* $\models$*as CNot* (*mset l* $-$ {#*a*#})
  ⟨*proof*⟩

lemma *is-unit-clause-some-in*: *is-unit-clause l M = Some a* $\Longrightarrow$ *a* $\in$ *set l*
  ⟨*proof*⟩

lemma *is-unit-clause-Nil*[*simp*]: *is-unit-clause* [] *M = None*
  ⟨*proof*⟩

Unit propagation for all clauses

Finding the first clause to propagate

fun *find-first-unit-clause* :: *'a literal list list* $\Rightarrow$ (*'a*, *'b*) *ann-lits*
  $\Rightarrow$ (*'a literal* $\times$ *'a literal list*) *option* **where**
find-first-unit-clause (*a* # *l*) *M* =
  (*case is-unit-clause a M of*
    None $\Rightarrow$ *find-first-unit-clause l M*
  | *Some L* $\Rightarrow$ *Some* (*L*, *a*)) |
find-first-unit-clause [] - = *None*

lemma *find-first-unit-clause-some*:
  find-first-unit-clause l M = Some (*a*, *c*)
  $\Longrightarrow$ *c* $\in$ *set l* $\wedge$ *M* $\models$*as CNot* (*mset c* $-$ {#*a*#}) $\wedge$ *undefined-lit M a* $\wedge$ *a* $\in$ *set c*
  ⟨*proof*⟩

lemma *propagate-is-unit-clause-not-None*:
  assumes *dist*: *distinct c* **and**
  M: *M* $\models$*as CNot* (*mset c* $-$ {#*a*#}) **and**
  undef: *undefined-lit M a* **and**
  ac: *a* $\in$ *set c*
  shows *is-unit-clause c M* $\neq$ *None*
⟨*proof*⟩

lemma *find-first-unit-clause-none*:
  distinct c $\Longrightarrow$ *c* $\in$ *set l* $\Longrightarrow$ *M* $\models$*as CNot* (*mset c* $-$ {#*a*#}) $\Longrightarrow$ *undefined-lit M a* $\Longrightarrow$ *a* $\in$ *set c*
  $\Longrightarrow$ *find-first-unit-clause l M* $\neq$ *None*
  ⟨*proof*⟩

Decide

fun *find-first-unused-var* :: *'a literal list list* $\Rightarrow$ *'a literal set* $\Rightarrow$ *'a literal option* **where**
find-first-unused-var (*a* # *l*) *M* =
  (*case List.find* (λ*lit. lit* $\notin$ *M* $\wedge$ $-$*lit* $\notin$ *M*) *a of*
    None $\Rightarrow$ *find-first-unused-var l M*
  | *Some a* $\Rightarrow$ *Some a*) |
find-first-unused-var [] - = *None*

lemma *find-none*[*iff*]:

List.find $(\lambda lit.\ lit \notin M \wedge -lit \notin M)$ *a* $=$ *None* $\longleftrightarrow$ *atm-of* ` *set a* $\subseteq$ *atm-of* ` *M*
⟨*proof*⟩

lemma *find-some*: *List.find* $(\lambda lit.\ lit \notin M \wedge -lit \notin M)$ *a* $=$ *Some b* $\Longrightarrow$ $b \in$ *set a* $\wedge\ b \notin M \wedge -b \notin M$
⟨*proof*⟩

lemma *find-first-unused-var-None*[*iff*]:
find-first-unused-var l M $=$ *None* $\longleftrightarrow$ $(\forall\, a \in$ *set l. atm-of* ` *set a* $\subseteq$ *atm-of* ` *M*)
⟨*proof*⟩

lemma *find-first-unused-var-Some-not-all-incl*:
assumes *find-first-unused-var l M* $=$ *Some c*
shows $\neg(\forall\, a \in$ *set l. atm-of* ` *set a* $\subseteq$ *atm-of* ` *M*)
⟨*proof*⟩

lemma *find-first-unused-var-Some*:
find-first-unused-var l M $=$ *Some a* $\Longrightarrow$ $(\exists\, m \in$ *set l.* $a \in$ *set m* $\wedge\ a \notin M \wedge -a \notin M)$
⟨*proof*⟩

lemma *find-first-unused-var-undefined*:
find-first-unused-var l (*lits-of-l Ms*) $=$ *Some a* $\Longrightarrow$ *undefined-lit Ms a*
⟨*proof*⟩

7.1.2 CDCL specific functions

Level

fun *maximum-level-code*:: *'a literal list* $\Rightarrow$ (*'a*, *'b*) *ann-lits* $\Rightarrow$ *nat*
where
maximum-level-code [] - $=$ *0* |
maximum-level-code (*L* # *Ls*) *M* $=$ *max* (*get-level M L*) (*maximum-level-code Ls M*)

lemma *maximum-level-code-eq-get-maximum-level*[*simp*]:
maximum-level-code D M $=$ *get-maximum-level M* (*mset D*)
⟨*proof*⟩

lemma [*code*]:
fixes *M* :: (*'a*, *'b*) *ann-lits*
shows *get-maximum-level M* (*mset D*) $=$ *maximum-level-code D M*
⟨*proof*⟩

Backjumping

fun *find-level-decomp* **where**
find-level-decomp M [] *D k* $=$ *None* |
find-level-decomp M (*L* # *Ls*) *D k* $=$
(*case* (*get-level M L*, *maximum-level-code* (*D* @ *Ls*) *M*) *of*
(*i*, *j*) $\Rightarrow$ *if* $i = k \wedge j < i$ *then Some* (*L*, *j*) *else find-level-decomp M Ls* (*L*#*D*) *k*
)

lemma *find-level-decomp-some*:
assumes *find-level-decomp M Ls D k* $=$ *Some* (*L*, *j*)
shows $L \in$ *set Ls* $\wedge$ *get-maximum-level M* (*mset* (*remove1 L* (*Ls* @ *D*))) $= j \wedge$ *get-level M L* $= k$
⟨*proof*⟩

lemma *find-level-decomp-none*:

assumes *find-level-decomp M Ls E k = None* **and** *mset (L#D) = mset (Ls @ E)*
shows ¬(*L* ∈ *set Ls* ∧ *get-maximum-level M (mset D) < k* ∧ *k = get-level M L*)
⟨*proof*⟩

fun *bt-cut* **where**
bt-cut i (Propagated - - # Ls) = bt-cut i Ls |
bt-cut i (Decided K # Ls) = (if count-decided Ls = i then Some (Decided K # Ls) else bt-cut i Ls) |
bt-cut i [] = None

lemma *bt-cut-some-decomp*:
assumes *no-dup M* **and** *bt-cut i M = Some M'*
shows ∃ *K M2 M1. M = M2 @ M'* ∧ *M' = Decided K # M1* ∧ *get-level M K = (i+1)*
⟨*proof*⟩

lemma *bt-cut-not-none*:
assumes *no-dup M* **and** *M = M2 @ Decided K # M'* **and** *get-level M K = (i+1)*
shows *bt-cut i M* ≠ *None*
⟨*proof*⟩

lemma *get-all-ann-decomposition-ex*:
∃ *N. (Decided K # M', N)* ∈ *set (get-all-ann-decomposition (M2@Decided K # M'))*
⟨*proof*⟩

lemma *bt-cut-in-get-all-ann-decomposition*:
assumes *no-dup M* **and** *bt-cut i M = Some M'*
shows ∃ *M2. (M', M2)* ∈ *set (get-all-ann-decomposition M)*
⟨*proof*⟩

fun *do-backtrack-step* **where**
do-backtrack-step (M, N, U, k, Some D) =
(*case find-level-decomp M D [] k of*
None ⇒ *(M, N, U, k, Some D)*
| *Some (L, j)* ⇒
(*case bt-cut j M of*
Some (Decided - # Ls) ⇒ *(Propagated L D # Ls, N, D # U, j, None)*
| - ⇒ *(M, N, U, k, Some D)*)
) |
do-backtrack-step S = S

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

7.1.3 Simple Implementation of DPLL

Combining the propagate and decide: a DPLL step

definition *DPLL-step* :: *int* $dpll_W$*-ann-lits* × *int literal list list*
⇒ *int* $dpll_W$*-ann-lits* × *int literal list list* **where**
DPLL-step = (λ*(Ms, N)*.
(*case find-first-unit-clause N Ms of*
Some (L, -) ⇒ *(Propagated L () # Ms, N)*
| - ⇒
if ∃ *C* ∈ *set N.* (∀ *c* ∈ *set C.* −*c* ∈ *lits-of-l Ms*)
then

(*case backtrack-split Ms of*
  (-, L # M) ⇒ (*Propagated* (− (*lit-of L*)) () # *M*, *N*)
| (-, -) ⇒ (*Ms*, *N*)
)
else
(*case find-first-unused-var N* (*lits-of-l Ms*) *of*
  Some a ⇒ (*Decided a* # *Ms*, *N*)
| *None* ⇒ (*Ms*, *N*))))

Example of propagation:

value *DPLL-step* ([*Decided* (*Neg 1*)], [[*Pos* (*1*::*int*), *Neg 2*]])

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation *toS* ≡ λ(*Ms*::(*int*, *unit*) *ann-lits*)
  (*N*:: *int literal list list*). (*Ms*, *mset* (*map mset N*))
abbreviation *toS′* ≡ λ(*Ms*::(*int*, *unit*) *ann-lits*,
  N:: *int literal list list*). (*Ms*, *mset* (*map mset N*))

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-dpll*$_W$*-step*:
  assumes *step*: (*Ms′*, *N′*) = *DPLL-step* (*Ms*, *N*)
  and *neq*: (*Ms*, *N*) ≠ (*Ms′*, *N′*)
  shows *dpll*$_W$ (*toS Ms N*) (*toS Ms′ N′*)
⟨*proof*⟩

lemma *DPLL-step-stuck-final-state*:
  assumes *step*: (*Ms*, *N*) = *DPLL-step* (*Ms*, *N*)
  shows *conclusive-dpll*$_W$*-state* (*toS Ms N*)
⟨*proof*⟩

Adding invariants

Invariant tested in the function **function** *DPLL-ci* :: *int dpll*$_W$*-ann-lits* ⇒ *int literal list list* ⇒ *int dpll*$_W$*-ann-lits* × *int literal list list* **where**
DPLL-ci Ms N =
  (*if* ¬*dpll*$_W$*-all-inv* (*Ms*, *mset* (*map mset N*))
  then (*Ms*, *N*)
  else
   let (*Ms′*, *N′*) = *DPLL-step* (*Ms*, *N*) *in*
   if (*Ms′*, *N′*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci Ms′ N*)
  ⟨*proof*⟩
termination
⟨*proof*⟩

No invariant tested **function** (*domintros*) *DPLL-part*:: *int dpll*$_W$*-ann-lits* ⇒ *int literal list list* ⇒ *int dpll*$_W$*-ann-lits* × *int literal list list* **where**
DPLL-part Ms N =
  (*let* (*Ms′*, *N′*) = *DPLL-step* (*Ms*, *N*) *in*
   if (*Ms′*, *N′*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-part Ms′ N*)
  ⟨*proof*⟩

lemma *snd-DPLL-step*[*simp*]:
  snd (*DPLL-step* (*Ms*, *N*)) = *N*
  ⟨*proof*⟩

lemma $dpll_W$*-all-inv-implieS-2-eq3-and-dom*:
assumes $dpll_W$*-all-inv* (*Ms*, *mset* (*map mset N*))
shows *DPLL-ci Ms N* = *DPLL-part Ms N* ∧ *DPLL-part-dom* (*Ms*, *N*)
⟨*proof*⟩

lemma *DPLL-ci-*$dpll_W$*-rtranclp*:
assumes *DPLL-ci Ms N* = (*Ms'*, *N'*)
shows $dpll_W^{**}$ (*toS Ms N*) (*toS Ms' N*)
⟨*proof*⟩

lemma $dpll_W$*-all-inv-*$dpll_W$*-tranclp-irrefl*:
assumes $dpll_W$*-all-inv* (*Ms*, *N*)
and $dpll_W^{++}$ (*Ms*, *N*) (*Ms*, *N*)
shows *False*
⟨*proof*⟩

lemma *DPLL-ci-final-state*:
assumes *step*: *DPLL-ci Ms N* = (*Ms*, *N*)
and *inv*: $dpll_W$*-all-inv* (*toS Ms N*)
shows *conclusive-*$dpll_W$*-state* (*toS Ms N*)
⟨*proof*⟩

lemma *DPLL-step-obtains*:
obtains *Ms'* **where** (*Ms'*, *N*) = *DPLL-step* (*Ms*, *N*)
⟨*proof*⟩

lemma *DPLL-ci-obtains*:
obtains *Ms'* **where** (*Ms'*, *N*) = *DPLL-ci Ms N*
⟨*proof*⟩

lemma *DPLL-ci-no-more-step*:
assumes *step*: *DPLL-ci Ms N* = (*Ms'*, *N'*)
shows *DPLL-ci Ms' N'* = (*Ms'*, *N'*)
⟨*proof*⟩

lemma *DPLL-part-*$dpll_W$*-all-inv-final*:
fixes *M Ms'*:: (*int*, *unit*) *ann-lits* **and**
N :: *int literal list list*
assumes *inv*: $dpll_W$*-all-inv* (*Ms*, *mset* (*map mset N*))
and *MsN*: *DPLL-part Ms N* = (*Ms'*, *N*)
shows *conclusive-*$dpll_W$*-state* (*toS Ms' N*) ∧ $dpll_W^{**}$ (*toS Ms N*) (*toS Ms' N*)
⟨*proof*⟩

Embedding the invariant into the type

Defining the type **typedef** $dpll_W$*-state* =
{(*M*::(*int*, *unit*) *ann-lits*, *N*::*int literal list list*).
$dpll_W$*-all-inv* (*toS M N*)}
morphisms *rough-state-of state-of*
⟨*proof*⟩

lemma

DPLL-part-dom ([], *N*)
⟨*proof*⟩

Some type classes **instantiation** $dpll_W$*-state* :: *equal*
begin
definition *equal-*$dpll_W$*-state* :: $dpll_W$*-state* ⇒ $dpll_W$*-state* ⇒ *bool* **where**
equal-$dpll_W$*-state* *S* *S′* = (*rough-state-of* *S* = *rough-state-of* *S′*)
instance
⟨*proof*⟩
end

DPLL **definition** *DPLL-step′* :: $dpll_W$*-state* ⇒ $dpll_W$*-state* **where**
DPLL-step′ *S* = *state-of* (*DPLL-step* (*rough-state-of* *S*))

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

lemma *DPLL-step-*$dpll_W$*-conc-inv*:
DPLL-step (*rough-state-of* *S*) ∈ {(*M*, *N*). $dpll_W$*-all-inv* (*toS* *M* *N*)}
⟨*proof*⟩

lemma *rough-state-of-DPLL-step′-DPLL-step*[*simp*]:
rough-state-of (*DPLL-step′* *S*) = *DPLL-step* (*rough-state-of* *S*)
⟨*proof*⟩

function *DPLL-tot*:: $dpll_W$*-state* ⇒ $dpll_W$*-state* **where**
DPLL-tot *S* =
(*let* *S′* = *DPLL-step′* *S* *in*
if *S′* = *S* *then* *S* *else* *DPLL-tot* *S′*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma [*code*]:
DPLL-tot *S* =
(*let* *S′* = *DPLL-step′* *S* *in*
if *S′* = *S* *then* *S* *else* *DPLL-tot* *S′*) ⟨*proof*⟩

lemma *DPLL-tot-DPLL-step-DPLL-tot*[*simp*]: *DPLL-tot* (*DPLL-step′* *S*) = *DPLL-tot* *S*
⟨*proof*⟩

lemma *DOPLL-step′-DPLL-tot*[*simp*]:
DPLL-step′ (*DPLL-tot* *S*) = *DPLL-tot* *S*
⟨*proof*⟩

lemma *DPLL-tot-final-state*:
assumes *DPLL-tot* *S* = *S*
shows *conclusive-*$dpll_W$*-state* (*toS′* (*rough-state-of* *S*))
⟨*proof*⟩

lemma *DPLL-tot-star*:
assumes *rough-state-of* (*DPLL-tot* *S*) = *S′*
shows $dpll_W{}^{**}$ (*toS′* (*rough-state-of* *S*)) (*toS′* *S′*)
⟨*proof*⟩

lemma *rough-state-of-rough-state-of-Nil*[*simp*]:
  rough-state-of (*state-of* ([], *N*)) = ([], *N*)
  ⟨*proof*⟩

Theorem of correctness

lemma *DPLL-tot-correct*:
  assumes *rough-state-of* (*DPLL-tot* (*state-of* (([], *N*)))) = (*M*, *N'*)
  and (*M'*, *N''*) = *toS'* (*M*, *N'*)
  shows *M'* ⊨*asm* *N''* ⟷ *satisfiable* (*set-mset* *N''*)
⟨*proof*⟩

Code export

A conversion to *DPLL-W-Implementation.dpll$_W$-state* **definition** *Con* :: (*int*, *unit*) *ann-lits* × *int literal list list*
  ⇒ *dpll$_W$-state* **where**
  Con xs = *state-of* (*if dpll$_W$-all-inv* (*toS* (*fst xs*) (*snd xs*)) *then xs else* ([], []))
lemma [*code abstype*]:
  Con (*rough-state-of S*) = *S*
  ⟨*proof*⟩

  declare *rough-state-of-DPLL-step'-DPLL-step*[*code abstract*]

lemma *Con-DPLL-step-rough-state-of-state-of*[*simp*]:
  Con (*DPLL-step* (*rough-state-of s*)) = *state-of* (*DPLL-step* (*rough-state-of s*))
  ⟨*proof*⟩

A slightly different version of *DPLL-tot* where the returned boolean indicates the result.

definition *DPLL-tot-rep* **where**
DPLL-tot-rep S =
  (*let* (*M*, *N*) = (*rough-state-of* (*DPLL-tot S*)) *in* (∀ *A* ∈ *set N*. (∃ *a*∈*set A*. *a* ∈ *lits-of-l* (*M*)), *M*))

One version of the generated SML code is here, but not included in the generated document. The only differences are:

- export *'a literal* from the SML Module *Clausal-Logic*;
- export the constructor *Con* from *DPLL-W-Implementation*;
- export the *int* constructor from *Arith*.

  All these allows to test on the code on some examples.

end
theory *CDCL-W-Implementation*
imports *DPLL-CDCL-W-Implementation CDCL-W-Termination*
begin

7.1.4 List-based CDCL Implementation

We here have a very simple implementation of Weidenbach's CDCL, based on the same principle as the implementation of DPLL: iterating over-and-over on lists. We do not use any fancy data-structure (see the two-watched literals for a better suited data-structure).

The goal was (as for DPLL) to test the infrastructure and see if an important lemma was missing to prove the correctness and the termination of a simple implementation.

Types and Instantiation

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

type-synonym $'a$ $cdcl_W$*-mark* = $'a$ *clause*

type-synonym $'v$ $cdcl_W$*-ann-lit* = ($'v$, $'v$ $cdcl_W$*-mark*) *ann-lit*
type-synonym $'v$ $cdcl_W$*-ann-lits* = ($'v$, $'v$ $cdcl_W$*-mark*) *ann-lits*
type-synonym $'v$ $cdcl_W$*-state* =
$'v$ $cdcl_W$*-ann-lits* $\times$ $'v$ *clauses* $\times$ $'v$ *clauses* $\times$ *nat* $\times$ $'v$ *clause option*

abbreviation *raw-trail* :: $'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a$ **where**
raw-trail $\equiv (\lambda(M, \text{-}).\ M)$

abbreviation *raw-cons-trail* :: $'a \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$
where
raw-cons-trail $\equiv (\lambda L\ (M, S).\ (L\#M, S))$

abbreviation *raw-tl-trail* :: $'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$ **where**
raw-tl-trail $\equiv (\lambda(M, S).\ (tl\ M, S))$

abbreviation *raw-init-clss* :: $'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b$ **where**
raw-init-clss $\equiv \lambda(M, N, \text{-}).\ N$

abbreviation *raw-learned-clss* :: $'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c$ **where**
raw-learned-clss $\equiv \lambda(M, N, U, \text{-}).\ U$

abbreviation *raw-backtrack-lvl* :: $'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd$ **where**
raw-backtrack-lvl $\equiv \lambda(M, N, U, k, \text{-}).\ k$

abbreviation *raw-update-backtrack-lvl* :: $'d \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
raw-update-backtrack-lvl $\equiv \lambda k\ (M, N, U, \text{-}, S).\ \ (M, N, U, k, S)$

abbreviation *raw-conflicting* :: $'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$ **where**
raw-conflicting $\equiv \lambda(M, N, U, k, D).\ D$

abbreviation *raw-update-conflicting* :: $'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
raw-update-conflicting $\equiv \lambda S\ (M, N, U, k, \text{-}).\ \ (M, N, U, k, S)$

abbreviation *S0-*$cdcl_W$ $N \equiv (([], N, \{\#\}, 0, None) :: 'v\ cdcl_W\text{-}state)$

abbreviation *raw-add-learned-clss* **where**
raw-add-learned-clss $\equiv \lambda C\ (M, N, U, S).\ (M, N, \{\#C\#\} + U, S)$

abbreviation *raw-remove-cls* **where**
raw-remove-cls $\equiv \lambda C\ (M, N, U, S).\ (M, removeAll\text{-}mset\ C\ N, removeAll\text{-}mset\ C\ U, S)$

lemma *raw-trail-conv*: *raw-trail* $(M, N, U, k, D) = M$ **and**
clauses-conv: *raw-init-clss* $(M, N, U, k, D) = N$ **and**
raw-learned-clss-conv: *raw-learned-clss* $(M, N, U, k, D) = U$ **and**
raw-conflicting-conv: *raw-conflicting* $(M, N, U, k, D) = D$ **and**
raw-backtrack-lvl-conv: *raw-backtrack-lvl* $(M, N, U, k, D) = k$
⟨*proof*⟩

lemma *state-conv*:
  $S = (raw\text{-}trail\ S,\ raw\text{-}init\text{-}clss\ S,\ raw\text{-}learned\text{-}clss\ S,\ raw\text{-}backtrack\text{-}lvl\ S,\ raw\text{-}conflicting\ S)$
  ⟨*proof*⟩

interpretation $state_W$
  raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting
  $\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
  $\lambda(M,\ S).\ (tl\ M,\ S)$
  $\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ \{\#C\#\} + U,\ S)$
  $\lambda C\ (M,\ N,\ U,\ S).\ (M,\ removeAll\text{-}mset\ C\ N,\ removeAll\text{-}mset\ C\ U,\ S)$
  $\lambda(k{::}nat)\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
  $\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
  $\lambda N.\ ([],\ N,\ \{\#\},\ 0,\ None)$
  ⟨*proof*⟩

interpretation $conflict\text{-}driven\text{-}clause\text{-}learning_W$ *raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting*
  $\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
  $\lambda(M,\ S).\ (tl\ M,\ S)$
  $\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ \{\#C\#\} + U,\ S)$
  $\lambda C\ (M,\ N,\ U,\ S).\ (M,\ removeAll\text{-}mset\ C\ N,\ removeAll\text{-}mset\ C\ U,\ S)$
  $\lambda(k{::}nat)\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
  $\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
  $\lambda N.\ ([],\ N,\ \{\#\},\ 0,\ None)$
  ⟨*proof*⟩

declare *clauses-def*[*simp*]

lemma $cdcl_W$*-state-eq-equality*[*iff*]: $state\text{-}eq\ S\ T \longleftrightarrow S = T$
  ⟨*proof*⟩
declare *state-simp*[*simp del*]

lemma *reduce-trail-to-empty-trail*[*simp*]:
  $reduce\text{-}trail\text{-}to\ F\ ([],\ aa,\ ab,\ ac,\ b) = ([],\ aa,\ ab,\ ac,\ b)$
  ⟨*proof*⟩

lemma *raw-trail-reduce-trail-to-length-le*:
  assumes $length\ F > length\ (raw\text{-}trail\ S)$
  shows $raw\text{-}trail\ (reduce\text{-}trail\text{-}to\ F\ S) = []$
  ⟨*proof*⟩

lemma *reduce-trail-to*:
  $reduce\text{-}trail\text{-}to\ F\ S =$
    $((if\ length\ (raw\text{-}trail\ S) \geq length\ F$
    $then\ drop\ (length\ (raw\text{-}trail\ S) - length\ F)\ (raw\text{-}trail\ S)$
    $else\ []),\ raw\text{-}init\text{-}clss\ S,\ raw\text{-}learned\text{-}clss\ S,\ raw\text{-}backtrack\text{-}lvl\ S,\ raw\text{-}conflicting\ S)$
    (**is** $?S = \text{-}$)
⟨*proof*⟩

7.1.5 CDCL Implementation

Definition of the rules

Types **lemma** *true-raw-init-clss-remdups*[*simp*]:
  $I \models s\ (mset \circ remdups)\ `\ N \longleftrightarrow I \models s\ mset\ `\ N$

⟨*proof*⟩

lemma *satisfiable-mset-remdups*[*simp*]:
satisfiable ((*mset* ∘ *remdups*) ‘ *N*) ⟷ *satisfiable* (*mset* ‘ *N*)
⟨*proof*⟩

type-synonym $'v\ cdcl_W\text{-}state\text{-}inv\text{-}st = ('v, 'v\ literal\ list)\ ann\text{-}lit\ list \times$
$'v\ literal\ list\ list \times 'v\ literal\ list\ list \times nat \times 'v\ literal\ list\ option$

We need some functions to convert between our abstract state $'v\ cdcl_W\text{-}state$ and the concrete state $'v\ cdcl_W\text{-}state\text{-}inv\text{-}st$.

fun *convert* :: ('*a*, '*c list*) *ann-lit* ⇒ ('*a*, '*c multiset*) *ann-lit* **where**
convert (*Propagated L C*) = *Propagated L* (*mset C*) |
convert (*Decided K*) = *Decided K*

abbreviation *convertC* :: '*a list option* ⇒ '*a multiset option* **where**
convertC ≡ *map-option mset*

lemma *convert-Propagated*[*elim*!]:
$convert\ z = Propagated\ L\ C \Longrightarrow (\exists\ C'.\ z = Propagated\ L\ C' \wedge C = mset\ C')$
⟨*proof*⟩

lemma *is-decided-convert*[*simp*]: *is-decided* (*convert x*) = *is-decided x*
⟨*proof*⟩

lemma *get-level-map-convert*[*simp*]:
get-level (*map convert M*) *x* = *get-level M x*
⟨*proof*⟩

lemma *get-maximum-level-map-convert*[*simp*]:
get-maximum-level (*map convert M*) *D* = *get-maximum-level M D*
⟨*proof*⟩

Conversion function

fun *toS* :: $'v\ cdcl_W\text{-}state\text{-}inv\text{-}st \Rightarrow 'v\ cdcl_W\text{-}state$ **where**
toS (*M*, *N*, *U*, *k*, *C*) = (*map convert M*, *mset* (*map mset N*), *mset* (*map mset U*), *k*, *convertC C*)

Definition an abstract type

typedef $'v\ cdcl_W\text{-}state\text{-}inv = \{S::'v\ cdcl_W\text{-}state\text{-}inv\text{-}st.\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ S)\}$
morphisms *rough-state-of state-of*
⟨*proof*⟩

instantiation $cdcl_W\text{-}state\text{-}inv$ :: (*type*) *equal*
begin
definition $equal\text{-}cdcl_W\text{-}state\text{-}inv :: 'v\ cdcl_W\text{-}state\text{-}inv \Rightarrow 'v\ cdcl_W\text{-}state\text{-}inv \Rightarrow bool$ **where**
$equal\text{-}cdcl_W\text{-}state\text{-}inv\ S\ S' = (rough\text{-}state\text{-}of\ S = rough\text{-}state\text{-}of\ S')$
instance
⟨*proof*⟩
end

lemma *lits-of-map-convert*[*simp*]: *lits-of-l* (*map convert M*) = *lits-of-l M*
⟨*proof*⟩

lemma *atm-lit-of-convert*[*simp*]:

lit-of (*convert* x) = *lit-of* x
⟨*proof*⟩

lemma *undefined-lit-map-convert*[*iff*]:
undefined-lit (*map convert* M) $L \longleftrightarrow$ *undefined-lit* M L
⟨*proof*⟩

lemma *true-annot-map-convert*[*simp*]: *map convert* $M \models a$ $N \longleftrightarrow M \models a$ N
⟨*proof*⟩

lemma *true-annots-map-convert*[*simp*]: *map convert* $M \models as$ $N \longleftrightarrow M \models as$ N
⟨*proof*⟩

lemmas *propagateE*
lemma *find-first-unit-clause-some-is-propagate*:
assumes H: *find-first-unit-clause* (N @ U) M = *Some* (L, C)
shows *propagate* (*toS* (M, N, U, k, *None*)) (*toS* (*Propagated* L C # M, N, U, k, *None*))
⟨*proof*⟩

The Transitions

Propagate **definition** *do-propagate-step* **where**
do-propagate-step S =
(*case* S *of*
(M, N, U, k, *None*) ⇒
(*case find-first-unit-clause* (N @ U) M *of*
Some (L, C) ⇒ (*Propagated* L C # M, N, U, k, *None*)
| *None* ⇒ (M, N, U, k, *None*))
| S ⇒ S)

lemma *do-propgate-step*:
do-propagate-step $S \neq S \Longrightarrow$ *propagate* (*toS* S) (*toS* (*do-propagate-step* S))
⟨*proof*⟩

lemma *do-propagate-step-option*[*simp*]:
raw-conflicting $S \neq$ *None* $\Longrightarrow$ *do-propagate-step* $S = S$
⟨*proof*⟩

lemma *do-propagate-step-no-step*:
assumes *dist*: ∀ *c*∈*set* (*raw-init-clss* S @ *raw-learned-clss* S). *distinct* c **and**
prop-step: *do-propagate-step* $S = S$
shows *no-step propagate* (*toS* S)
⟨*proof*⟩

Conflict **fun** *find-conflict* **where**
find-conflict M [] = *None* |
find-conflict M (N # *Ns*) = (*if* ($\forall c \in$ *set* N. $-c \in$ *lits-of-l* M) *then Some* N *else find-conflict* M *Ns*)

lemma *find-conflict-Some*:
find-conflict M *Ns* = *Some* $N \Longrightarrow N \in$ *set Ns* $\wedge M \models as$ *CNot* (*mset* N)
⟨*proof*⟩

lemma *find-conflict-None*:
find-conflict M *Ns* = *None* $\longleftrightarrow$ ($\forall N \in$ *set Ns*. $\neg M \models as$ *CNot* (*mset* N))
⟨*proof*⟩

lemma *find-conflict-None-no-confl*:
$\quad$ *find-conflict* M $(N@U) = None \longleftrightarrow$ *no-step conflict* (*toS* $(M, N, U, k, None)$)
$\quad\langle proof\rangle$

definition *do-conflict-step* **where**
do-conflict-step $S =$
$\quad$(*case* S *of*
$\quad\quad(M, N, U, k, None) \Rightarrow$
$\quad\quad\quad$(*case find-conflict* M $(N$ @ $U)$ *of*
$\quad\quad\quad\quad$*Some* $a \Rightarrow (M, N, U, k, Some\ a)$
$\quad\quad\quad| None \Rightarrow (M, N, U, k, None))$
$\quad| S \Rightarrow S)$

lemma *do-conflict-step*:
$\quad$ *do-conflict-step* $S \neq S \Longrightarrow$ *conflict* (*toS* S) (*toS* (*do-conflict-step* S))
$\quad\langle proof\rangle$

lemma *do-conflict-step-no-step*:
$\quad$ *do-conflict-step* $S = S \Longrightarrow$ *no-step conflict* (*toS* S)
$\quad\langle proof\rangle$

lemma *do-conflict-step-option*[*simp*]:
$\quad$ *raw-conflicting* $S \neq None \Longrightarrow$ *do-conflict-step* $S = S$
$\quad\langle proof\rangle$

lemma *do-conflict-step-raw-conflicting*[*dest*]:
$\quad$ *do-conflict-step* $S \neq S \Longrightarrow$ *raw-conflicting* (*do-conflict-step* S) $\neq None$
$\quad\langle proof\rangle$

definition *do-cp-step* **where**
do-cp-step $S =$
$\quad$(*do-propagate-step o do-conflict-step*) S

lemma *cp-step-is-cdcl$_W$-cp*:
$\quad$**assumes** H: *do-cp-step* $S \neq S$
$\quad$**shows** *cdcl$_W$-cp* (*toS* S) (*toS* (*do-cp-step* S))
$\langle proof\rangle$

lemma *do-cp-step-eq-no-prop-no-confl*:
$\quad$ *do-cp-step* $S = S \Longrightarrow$ *do-conflict-step* $S = S \land$ *do-propagate-step* $S = S$
$\quad\langle proof\rangle$

lemma *no-cdcl$_W$-cp-iff-no-propagate-no-conflict*:
$\quad$ *no-step cdcl$_W$-cp* $S \longleftrightarrow$ *no-step propagate* $S \land$ *no-step conflict* S
$\quad\langle proof\rangle$

lemma *do-cp-step-eq-no-step*:
$\quad$**assumes** H: *do-cp-step* $S = S$ **and** $\forall\, c \in$ *set* (*raw-init-clss* S @ *raw-learned-clss* S). *distinct* c
$\quad$**shows** *no-step cdcl$_W$-cp* (*toS* S)
$\quad\langle proof\rangle$

lemma *cdcl$_W$-cp-cdcl$_W$-st*: *cdcl$_W$-cp* $S\ S' \Longrightarrow$ *cdcl$_W$*** $S\ S'$
$\quad\langle proof\rangle$

lemma *cdcl$_W$-all-struct-inv-rough-state*[*simp*]: *cdcl$_W$-all-struct-inv* (*toS* (*rough-state-of* S))
$\quad\langle proof\rangle$

lemma [*simp*]: $cdcl_W$*-all-struct-inv* (*toS S*) $\Longrightarrow$ *rough-state-of* (*state-of S*) = *S*
⟨*proof*⟩

lemma *rough-state-of-state-of-do-cp-step*[*simp*]:
rough-state-of (*state-of* (*do-cp-step* (*rough-state-of S*))) = *do-cp-step* (*rough-state-of S*)
⟨*proof*⟩

Skip **fun** *do-skip-step* :: *'v* $cdcl_W$*-state-inv-st* ⇒ *'v* $cdcl_W$*-state-inv-st* **where**
do-skip-step (*Propagated L C # Ls*,*N*,*U*,*k*, *Some D*) =
(*if* −*L* ∉ *set D* ∧ *D* ≠ []
then (*Ls*, *N*, *U*, *k*, *Some D*)
else (*Propagated L C #Ls*, *N*, *U*, *k*, *Some D*)) |
do-skip-step S = *S*

lemma *do-skip-step*:
do-skip-step S ≠ *S* $\Longrightarrow$ *skip* (*toS S*) (*toS* (*do-skip-step S*))
⟨*proof*⟩

lemma *do-skip-step-no*:
do-skip-step S = *S* $\Longrightarrow$ *no-step skip* (*toS S*)
⟨*proof*⟩

lemma *do-skip-step-raw-trail-is-None*[*iff*]:
do-skip-step S = (*a*, *b*, *c*, *d*, *None*) ⟷ *S* = (*a*, *b*, *c*, *d*, *None*)
⟨*proof*⟩

Resolve **fun** *maximum-level-code*:: *'a literal list* ⇒ (*'a*, *'a literal list*) *ann-lit list* ⇒ *nat*
where
maximum-level-code [] - = *0* |
maximum-level-code (*L # Ls*) *M* = *max* (*get-level M L*) (*maximum-level-code Ls M*)

lemma *maximum-level-code-eq-get-maximum-level*[*code*, *simp*]:
maximum-level-code D M = *get-maximum-level M* (*mset D*)
⟨*proof*⟩

fun *do-resolve-step* :: *'v* $cdcl_W$*-state-inv-st* ⇒ *'v* $cdcl_W$*-state-inv-st* **where**
do-resolve-step (*Propagated L C # Ls*, *N*, *U*, *k*, *Some D*) =
(*if* −*L* ∈ *set D* ∧ *maximum-level-code* (*remove1* (−*L*) *D*) (*Propagated L C # Ls*) = *k*
then (*Ls*, *N*, *U*, *k*, *Some* (*remdups* (*remove1 L C @ remove1* (−*L*) *D*)))
else (*Propagated L C # Ls*, *N*, *U*, *k*, *Some D*)) |
do-resolve-step S = *S*

lemma *do-resolve-step*:
$cdcl_W$*-all-struct-inv* (*toS S*) $\Longrightarrow$ *do-resolve-step S* ≠ *S*
$\Longrightarrow$ *resolve* (*toS S*) (*toS* (*do-resolve-step S*))
⟨*proof*⟩

lemma *do-resolve-step-no*:
do-resolve-step S = *S* $\Longrightarrow$ *no-step resolve* (*toS S*)
⟨*proof*⟩

lemma *rough-state-of-state-of-resolve*[*simp*]:
$cdcl_W$*-all-struct-inv* (*toS S*) $\Longrightarrow$ *rough-state-of* (*state-of* (*do-resolve-step S*)) = *do-resolve-step S*
⟨*proof*⟩

lemma *do-resolve-step-raw-trail-is-None*[*iff*]:
do-resolve-step $S = (a,\ b,\ c,\ d,\ None) \longleftrightarrow S = (a,\ b,\ c,\ d,\ None)$
⟨*proof*⟩

Backjumping

lemma *get-all-ann-decomposition-map-convert*:
(*get-all-ann-decomposition* (*map convert* M)) =
map ($\lambda(a,\ b)$. (*map convert* a, *map convert* b)) (*get-all-ann-decomposition* M)
⟨*proof*⟩

lemma *do-backtrack-step*:
assumes
db: *do-backtrack-step* $S \neq S$ **and**
inv: $cdcl_W$*-all-struct-inv* (*toS* S)
shows *backtrack* (*toS* S) (*toS* (*do-backtrack-step* S))
⟨*proof*⟩

lemma *map-eq-list-length*:
map $f\ L = L' \Longrightarrow$ *length* L = *length* L'
⟨*proof*⟩

lemma *map-mmset-of-mlit-eq-cons*:
assumes *map convert* $M = a\ @\ c$
obtains $a'\ c'$ **where**
$M = a'\ @\ c'$ **and**
a = *map convert* a' **and**
c = *map convert* c'
⟨*proof*⟩

lemma *Decided-convert-iff*:
Decided K = *convert za* $\longleftrightarrow$ *za* = *Decided* K
⟨*proof*⟩

lemma *do-backtrack-step-no*:
assumes
db: *do-backtrack-step* $S = S$ **and**
inv: $cdcl_W$*-all-struct-inv* (*toS* S)
shows *no-step backtrack* (*toS* S)
⟨*proof*⟩

lemma *rough-state-of-state-of-backtrack*[*simp*]:
assumes *inv*: $cdcl_W$*-all-struct-inv* (*toS* S)
shows *rough-state-of* (*state-of* (*do-backtrack-step* S))= *do-backtrack-step* S
⟨*proof*⟩

Decide

fun *do-decide-step* **where**
do-decide-step (M, N, U, k, *None*) =
(*case find-first-unused-var* N (*lits-of-l* M) *of*
None ⇒ (M, N, U, k, *None*)
| *Some* L ⇒ (*Decided* L # M, N, U, k+*1*, *None*)) |
do-decide-step $S = S$

lemma *do-decide-step*:
do-decide-step $S \neq S \Longrightarrow$ *decide* (*toS* S) (*toS* (*do-decide-step* S))
⟨*proof*⟩

lemma *do-decide-step-no*:
  $do\text{-}decide\text{-}step\ S = S \Longrightarrow no\text{-}step\ decide\ (toS\ S)$
  ⟨*proof*⟩

lemma *rough-state-of-state-of-do-decide-step*[*simp*]:
  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ S) \Longrightarrow rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}decide\text{-}step\ S)) = do\text{-}decide\text{-}step\ S$
⟨*proof*⟩

lemma *rough-state-of-state-of-do-skip-step*[*simp*]:
  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ S) \Longrightarrow rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}skip\text{-}step\ S)) = do\text{-}skip\text{-}step\ S$
  ⟨*proof*⟩

Code generation

Type definition There are two invariants: one while applying conflict and propagate and one for the other rules

declare *rough-state-of-inverse*[*simp add*]
definition *Con* **where**
  $Con\ xs = state\text{-}of\ (if\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ (fst\ xs,\ snd\ xs))\ then\ xs$
  $else\ ([],\ [],\ [],\ 0,\ None))$

lemma [*code abstype*]:
 $Con\ (rough\text{-}state\text{-}of\ S) = S$
  ⟨*proof*⟩

definition *do-cp-step′* **where**
$do\text{-}cp\text{-}step'\ S = state\text{-}of\ (do\text{-}cp\text{-}step\ (rough\text{-}state\text{-}of\ S))$

typedef $'v\ cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state =$
  $\{S{::}\ 'v\ cdcl_W\text{-}state\text{-}inv\text{-}st.\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ S)$
    $\wedge\ cdcl_W\text{-}stgy^{**}\ (S0\text{-}cdcl_W\ (raw\text{-}init\text{-}clss\ (toS\ S)))\ (toS\ S)\}$
  morphisms *rough-state-from-init-state-of state-from-init-state-of*
⟨*proof*⟩

instantiation $cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state :: (type)\ equal$
begin
definition $equal\text{-}cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state :: 'v\ cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state \Rightarrow$
  $'v\ cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state \Rightarrow bool$ **where**
 $equal\text{-}cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state\ S\ S' \longleftrightarrow$
   $(rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S = rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S')$
instance
  ⟨*proof*⟩
end

definition *ConI* **where**
  $ConI\ S = state\text{-}from\text{-}init\text{-}state\text{-}of\ (if\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ (fst\ S,\ snd\ S))$
    $\wedge\ cdcl_W\text{-}stgy^{**}\ (S0\text{-}cdcl_W\ (raw\text{-}init\text{-}clss\ (toS\ S)))\ (toS\ S)\ then\ S\ else\ ([],\ [],\ [],\ 0,\ None))$

lemma [*code abstype*]:
  $ConI\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S) = S$
  ⟨*proof*⟩

definition $id\text{-}of\text{-}I\text{-}to{::}\ 'v\ cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state \Rightarrow 'v\ cdcl_W\text{-}state\text{-}inv$ **where**

id-of-I-to S = *state-of* (*rough-state-from-init-state-of* S)

lemma [*code abstract*]:
  rough-state-of (*id-of-I-to* S) = *rough-state-from-init-state-of* S
  ⟨*proof*⟩

Conflict and Propagate **function** *do-full1-cp-step* :: $'v$ $cdcl_W$*-state-inv* ⇒ $'v$ $cdcl_W$*-state-inv*
where
do-full1-cp-step S =
  (*let* S' = *do-cp-step'* S *in*
   if $S = S'$ *then* S *else* *do-full1-cp-step* S')
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *do-full1-cp-step-fix-point-of-do-full1-cp-step*:
  do-cp-step(*rough-state-of* (*do-full1-cp-step* S)) = (*rough-state-of* (*do-full1-cp-step* S))
  ⟨*proof*⟩

lemma *in-clauses-rough-state-of-is-distinct*:
  c∈*set* (*raw-init-clss* (*rough-state-of* S) @ *raw-learned-clss* (*rough-state-of* S)) ⟹ *distinct* c
  ⟨*proof*⟩

lemma *do-full1-cp-step-full*:
  full $cdcl_W$*-cp* (*toS* (*rough-state-of* S))
    (*toS* (*rough-state-of* (*do-full1-cp-step* S)))
  ⟨*proof*⟩

lemma [*code abstract*]:
 rough-state-of (*do-cp-step'* S) = *do-cp-step* (*rough-state-of* S)
 ⟨*proof*⟩

The other rules **fun** *do-other-step* **where**
do-other-step S =
   (*let* T = *do-skip-step* S *in*
     if $T \neq S$
     then T
     else
       (*let* U = *do-resolve-step* T *in*
       if $U \neq T$
       then U *else*
       (*let* V = *do-backtrack-step* U *in*
       if $V \neq U$ *then* V *else* *do-decide-step* V)))

lemma *do-other-step*:
  assumes *inv*: $cdcl_W$*-all-struct-inv* (*toS* S) **and**
  st: *do-other-step* $S \neq S$
  shows $cdcl_W$*-o* (*toS* S) (*toS* (*do-other-step* S))
  ⟨*proof*⟩

lemma *do-other-step-no*:
  assumes *inv*: $cdcl_W$*-all-struct-inv* (*toS* S) **and**
  st: *do-other-step* $S = S$
  shows *no-step* $cdcl_W$*-o* (*toS* S)
  ⟨*proof*⟩

lemma *rough-state-of-state-of-do-other-step*[*simp*]:
rough-state-of (*state-of* (*do-other-step* (*rough-state-of S*))) = *do-other-step* (*rough-state-of S*)
⟨*proof*⟩

definition *do-other-step′* **where**
do-other-step′ S =
state-of (*do-other-step* (*rough-state-of S*))

lemma *rough-state-of-do-other-step′*[*code abstract*]:
rough-state-of (*do-other-step′ S*) = *do-other-step* (*rough-state-of S*)
⟨*proof*⟩

definition *do-cdcl*$_W$*-stgy-step* **where**
do-cdcl$_W$*-stgy-step S* =
(*let T* = *do-full1-cp-step S in*
if T ≠ *S*
then T
else
(*let U* = (*do-other-step′ T*) *in*
(*do-full1-cp-step U*)))

definition *do-cdcl*$_W$*-stgy-step′* **where**
do-cdcl$_W$*-stgy-step′ S* = *state-from-init-state-of* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*id-of-I-to S*)))

lemma *toS-do-full1-cp-step-not-eq*: *do-full1-cp-step S* ≠ *S* ⟹
toS (*rough-state-of S*) ≠ *toS* (*rough-state-of* (*do-full1-cp-step S*))
⟨*proof*⟩

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps*[*simp del*]

Correction of the transformation

lemma *do-cdcl*$_W$*-stgy-step*:
assumes *do-cdcl*$_W$*-stgy-step S* ≠ *S*
shows *cdcl*$_W$*-stgy* (*toS* (*rough-state-of S*)) (*toS* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step S*)))
⟨*proof*⟩

lemma *length-raw-trail-toS*[*simp*]:
length (*raw-trail* (*toS S*)) = *length* (*raw-trail S*)
⟨*proof*⟩

lemma *raw-conflicting-noTrue-iff-toS*[*simp*]:
raw-conflicting (*toS S*) ≠ *None* ⟷ *raw-conflicting S* ≠ *None*
⟨*proof*⟩

lemma *raw-trail-toS-neq-imp-raw-trail-neq*:
raw-trail (*toS S*) ≠ *raw-trail* (*toS S′*) ⟹ *raw-trail S* ≠ *raw-trail S′*
⟨*proof*⟩

lemma *do-skip-step-raw-trail-changed-or-conflict*:
assumes *d*: *do-other-step S* ≠ *S*
and *inv*: *cdcl*$_W$*-all-struct-inv* (*toS S*)
shows *raw-trail S* ≠ *raw-trail* (*do-other-step S*)
⟨*proof*⟩

lemma *do-full1-cp-step-induct*:

$(\bigwedge S.\ (S \neq\ do\text{-}cp\text{-}step'\ S \Longrightarrow P\ (do\text{-}cp\text{-}step'\ S)) \Longrightarrow P\ S) \Longrightarrow P\ a0$
⟨proof⟩

lemma *do-cp-step-neq-raw-trail-increase*:
$\exists c.\ raw\text{-}trail\ (do\text{-}cp\text{-}step\ S) = c\ @\ raw\text{-}trail\ \ S \land(\forall m \in set\ c.\ \neg\ is\text{-}decided\ m)$
⟨proof⟩

lemma *do-full1-cp-step-neq-raw-trail-increase*:
$\exists c.\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S)) = c\ @\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)$
$\land (\forall m \in set\ c.\ \neg\ is\text{-}decided\ m)$
⟨proof⟩

lemma *do-cp-step-raw-conflicting*:
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) \neq None \Longrightarrow do\text{-}cp\text{-}step'\ S = S$
⟨proof⟩

lemma *do-full1-cp-step-raw-conflicting*:
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) \neq None \Longrightarrow do\text{-}full1\text{-}cp\text{-}step\ S = S$
⟨proof⟩

lemma *do-decide-step-not-raw-conflicting-one-more-decide*:
assumes
$raw\text{-}conflicting\ S = None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $Suc\ (length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ S)))$
$= length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
⟨proof⟩

lemma *do-decide-step-not-raw-conflicting-one-more-decide-bt*:
assumes $raw\text{-}conflicting\ S \neq None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ S)) < length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
⟨proof⟩

lemma *count-decided-raw-trail-toS*:
$count\text{-}decided\ (raw\text{-}trail\ (toS\ S)) =\ count\text{-}decided\ (raw\text{-}trail\ S)$
⟨proof⟩

lemma *do-other-step-not-raw-conflicting-one-more-decide-bt*:
assumes
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) \neq None$ **and**
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S)) = None$ **and**
$do\text{-}other\text{-}step'\ S \neq S$
shows $count\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ S))$
$> count\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S)))$
⟨proof⟩

lemma *do-other-step-not-raw-conflicting-one-more-decide*:
assumes $raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) = None$ **and**
$do\text{-}other\text{-}step'\ S \neq S$
shows $1 + length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)))$
$= length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S))))$
⟨proof⟩

lemma *rough-state-of-state-of-do-skip-step-rough-state-of*[*simp*]:
$rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}skip\text{-}step\ (rough\text{-}state\text{-}of\ S))) = do\text{-}skip\text{-}step\ (rough\text{-}state\text{-}of\ S)$

$\langle proof \rangle$

lemma *raw-conflicting-do-resolve-step-iff* [*iff*]:
raw-conflicting (*do-resolve-step S*) = *None* $\longleftrightarrow$ *raw-conflicting S* = *None*
$\langle proof \rangle$

lemma *raw-conflicting-do-skip-step-iff* [*iff*]:
raw-conflicting (*do-skip-step S*) = *None* $\longleftrightarrow$ *raw-conflicting S* = *None*
$\langle proof \rangle$

lemma *raw-conflicting-do-decide-step-iff* [*iff*]:
raw-conflicting (*do-decide-step S*) = *None* $\longleftrightarrow$ *raw-conflicting S* = *None*
$\langle proof \rangle$

lemma *raw-conflicting-do-backtrack-step-imp*[*simp*]:
do-backtrack-step S $\neq$ *S* $\Longrightarrow$ *raw-conflicting* (*do-backtrack-step S*) = *None*
$\langle proof \rangle$

lemma *do-skip-step-eq-iff-raw-trail-eq*:
do-skip-step S = *S* $\longleftrightarrow$ *raw-trail* (*do-skip-step S*) = *raw-trail S*
$\langle proof \rangle$

lemma *do-decide-step-eq-iff-raw-trail-eq*:
do-decide-step S = *S* $\longleftrightarrow$ *raw-trail* (*do-decide-step S*) = *raw-trail S*
$\langle proof \rangle$

lemma *do-backtrack-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail S*)
shows *do-backtrack-step S* = *S* $\longleftrightarrow$ *raw-trail* (*do-backtrack-step S*) = *raw-trail S*
$\langle proof \rangle$

lemma *do-resolve-step-eq-iff-raw-trail-eq*:
do-resolve-step S = *S* $\longleftrightarrow$ *raw-trail* (*do-resolve-step S*) = *raw-trail S*
$\langle proof \rangle$

lemma *do-other-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail S*)
shows *raw-trail* (*do-other-step S*) = *raw-trail S* $\longleftrightarrow$ *do-other-step S* = *S*
$\langle proof \rangle$

lemma *do-full1-cp-step-do-other-step'-normal-form*[*dest!*]:
assumes *H*: *do-full1-cp-step* (*do-other-step' S*) = *S*
shows *do-other-step' S* = *S* $\wedge$ *do-full1-cp-step S* = *S*
$\langle proof \rangle$

lemma *do-cdcl$_W$-stgy-step-no*:
assumes *S*: *do-cdcl$_W$-stgy-step S* = *S*
shows *no-step cdcl$_W$-stgy* (*toS* (*rough-state-of S*))
$\langle proof \rangle$

lemma *toS-rough-state-of-state-of-rough-state-from-init-state-of* [*simp*]:
toS (*rough-state-of* (*state-of* (*rough-state-from-init-state-of S*)))
= *toS* (*rough-state-from-init-state-of S*)
$\langle proof \rangle$

lemma $cdcl_W$*-cp-is-rtranclp-*$cdcl_W$: $cdcl_W\text{-}cp\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-is-rtranclp-*$cdcl_W$: $cdcl_W\text{-}cp^{**}\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
⟨*proof*⟩

lemma $cdcl_W$*-stgy-is-rtranclp-*$cdcl_W$:
$cdcl_W\text{-}stgy\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
⟨*proof*⟩

lemma $cdcl_W$*-stgy-init-raw-init-clss*:
$cdcl_W\text{-}stgy\ S\ T \Longrightarrow cdcl_W\text{-}M\text{-}level\text{-}inv\ S \Longrightarrow raw\text{-}init\text{-}clss\ S = raw\text{-}init\text{-}clss\ T$
⟨*proof*⟩

lemma *clauses-toS-rough-state-of-do-*$cdcl_W$*-stgy-step*[*simp*]:
$raw\text{-}init\text{-}clss\ (toS\ (rough\text{-}state\text{-}of\ (do\text{-}cdcl_W\text{-}stgy\text{-}step\ (state\text{-}of\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S)))))$
$= raw\text{-}init\text{-}clss\ (toS\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S))$ (**is** - $=$ $raw\text{-}init\text{-}clss\ (toS\ ?S)$)
⟨*proof*⟩

lemma *rough-state-from-init-state-of-do-*$cdcl_W$*-stgy-step*′[*code abstract*]:
$rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ (do\text{-}cdcl_W\text{-}stgy\text{-}step'\ S) =$
$rough\text{-}state\text{-}of\ (do\text{-}cdcl_W\text{-}stgy\text{-}step\ (id\text{-}of\text{-}I\text{-}to\ S))$
⟨*proof*⟩

All rules together **function** *do-all-*$cdcl_W$*-stgy* **where**
$do\text{-}all\text{-}cdcl_W\text{-}stgy\ S =$
$(let\ T = do\text{-}cdcl_W\text{-}stgy\text{-}step'\ S\ in$
$if\ T = S\ then\ S\ else\ do\text{-}all\text{-}cdcl_W\text{-}stgy\ T)$
⟨*proof*⟩
termination
⟨*proof*⟩

thm *do-all-*$cdcl_W$*-stgy.induct*
lemma *do-all-*$cdcl_W$*-stgy-induct*:
$(\bigwedge S.\ (do\text{-}cdcl_W\text{-}stgy\text{-}step'\ S \neq S \Longrightarrow P\ (do\text{-}cdcl_W\text{-}stgy\text{-}step'\ S)) \Longrightarrow P\ S) \Longrightarrow P\ a0$
⟨*proof*⟩

lemma *no-step-*$cdcl_W$*-stgy-*$cdcl_W$*-all*:
fixes S :: $'a\ cdcl_W\text{-}state\text{-}inv\text{-}from\text{-}init\text{-}state$
shows $no\text{-}step\ cdcl_W\text{-}stgy\ (toS\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ (do\text{-}all\text{-}cdcl_W\text{-}stgy\ S)))$
⟨*proof*⟩

lemma *do-all-*$cdcl_W$*-stgy-is-rtranclp-*$cdcl_W$*-stgy*:
$cdcl_W\text{-}stgy^{**}\ (toS\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S))$
$(toS\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ (do\text{-}all\text{-}cdcl_W\text{-}stgy\ S)))$
⟨*proof*⟩

Final theorem:

lemma *DPLL-tot-correct*:
assumes
r: $rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ (do\text{-}all\text{-}cdcl_W\text{-}stgy\ (state\text{-}from\text{-}init\text{-}state\text{-}of$
$(([],\ map\ remdups\ N,\ [],\ 0,\ None)))) = S$ **and**
S: $(M',\ N',\ U',\ k,\ E) = toS\ S$
shows $(E \neq Some\ \{\#\} \wedge satisfiable\ (set\ (map\ mset\ N)))$
$\vee\ (E = Some\ \{\#\} \wedge unsatisfiable\ (set\ (map\ mset\ N)))$

⟨*proof*⟩

The Code The SML code is skipped in the documentation, but stays to ensure that some version of the exported code is working. The only difference between the generated code and the one used here is the export of the constructor ConI.

end
theory *CDCL-Abstract-Clause-Representation*
imports *Main Partial-Clausal-Logic*
begin

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

7.1.6 Abstract Clause Representation

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

We assume the following:

- there is an equivalent to adding and removing a literal and to taking the union of clauses.

locale *raw-cls* =
  fixes
    mset-cls :: $'cls \Rightarrow 'v$ *clause*
begin
end

The two following locales are the *exact same* locale, but we need two different locales. Otherwise, instantiating *raw-clss* would lead to duplicate constants. (TODO: better idea?).

locale *abstract-with-index* =
  fixes
    get-lit :: $'a \Rightarrow 'it \Rightarrow 'conc$ *option* **and**
    convert-to-mset :: $'a \Rightarrow 'conc$ *multiset*
  assumes
    in-clss-mset-cls[*dest*]:
      get-lit Cs a = *Some e* $\Longrightarrow$ *e* $\in\#$ *convert-to-mset Cs* **and**
    in-mset-cls-exists-preimage:
      b $\in\#$ *convert-to-mset Cs* $\Longrightarrow \exists\, b'.$ *get-lit Cs* b' = *Some b*

locale *abstract-with-index2* =
  fixes
    get-lit :: $'a \Rightarrow 'it \Rightarrow 'conc$ *option* **and**
    convert-to-mset :: $'a \Rightarrow 'conc$ *multiset*
  assumes
    in-clss-mset-clss[*dest*]:
      get-lit Cs a = *Some e* $\Longrightarrow$ *e* $\in\#$ *convert-to-mset Cs* **and**
    in-mset-clss-exists-preimage:
      b $\in\#$ *convert-to-mset Cs* $\Longrightarrow \exists\, b'.$ *get-lit Cs* b' = *Some b*

locale *raw-clss* =
  abstract-with-index get-lit mset-cls +
  abstract-with-index2 get-cls mset-clss

for
  get-lit :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
  mset-cls :: *'cls* ⇒ *'v clause* **and**

  get-cls :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
  mset-clss:: *'clss* ⇒ *'cls multiset*
begin

definition *cls-lit* :: *'cls* ⇒ *'lit* ⇒ *'v literal* (**infix** ↓ *49*) **where**
C ↓ *a* ≡ *the* (*get-lit C a*)

definition *clss-cls* :: *'clss* ⇒ *'cls-it* ⇒ *'cls* (**infix** ⇓ *49*) **where**
C ⇓ *a* ≡ *the* (*get-cls C a*)

definition *in-cls* :: *'lit* ⇒ *'cls* ⇒ *bool* (**infix** ∈↓ *49*) **where**
a ∈↓ *Cs* ≡ *get-lit Cs a* ≠ *None*

definition *in-clss* :: *'cls-it* ⇒ *'clss* ⇒ *bool* (**infix** ∈⇓ *49*) **where**
a ∈⇓ *Cs* ≡ *get-cls Cs a* ≠ *None*

definition *raw-clss* **where**
raw-clss S ≡ *image-mset mset-cls* (*mset-clss S*)

end

experiment
begin
  fun *safe-nth* **where**
  safe-nth (*x* # -) *0* = *Some x* |
  safe-nth (- # *xs*) (*Suc n*) = *safe-nth xs n* |
  safe-nth [] - = *None*

  lemma *safe-nth-nth*: *n* < *length l* ⟹ *safe-nth l n* = *Some* (*nth l n*)
    ⟨*proof*⟩

  lemma *safe-nth-None*: *n* ≥ *length l* ⟹ *safe-nth l n* = *None*
    ⟨*proof*⟩

  lemma *safe-nth-Some-iff*: *safe-nth l n* = *Some m* ⟷ *n* < *length l* ∧ *m* = *nth l n*
    ⟨*proof*⟩

  lemma *safe-nth-None-iff*: *safe-nth l n* = *None* ⟷ *n* ≥ *length l*
    ⟨*proof*⟩

  interpretation *abstract-with-index*
    safe-nth
    mset
    ⟨*proof*⟩

  interpretation *abstract-with-index2*
    safe-nth
    mset
    ⟨*proof*⟩

  interpretation *list-cls*: *raw-clss*
    safe-nth mset

safe-nth mset
⟨*proof*⟩
end

end
theory *CDCL-W-Abstract-State*
imports *CDCL-Abstract-Clause-Representation CDCL-WNOT*

begin

7.2 Weidenbach's CDCL with Abstract Clause Representation

We first instantiate the locale of Weidenbach's locale. Then we define another abstract state: the goal of this state is to be used for implementations. We add more assumptions on the function about the state. For example *cons-trail* is restricted to undefined literals.

7.2.1 Instantiation of the Multiset Version

type-synonym $'v\ cdcl_W\text{-}mset = ('v,\ 'v\ clause)\ ann\text{-}lit\ list \times$
$'v\ clauses \times$
$'v\ clauses \times$
$nat \times\ 'v\ clause\ option$

We use definition, otherwise we could not use the simplification theorems we have already shown.

definition $trail :: 'v\ cdcl_W\text{-}mset \Rightarrow ('v,\ 'v\ clause)\ ann\text{-}lit\ list$ **where**
$trail \equiv \lambda(M,\ \text{-}).\ M$

definition $init\text{-}clss :: 'v\ cdcl_W\text{-}mset \Rightarrow\ 'v\ clauses$ **where**
$init\text{-}clss \equiv \lambda(\text{-},\ N,\ \text{-}).\ N$

definition $learned\text{-}clss :: 'v\ cdcl_W\text{-}mset \Rightarrow\ 'v\ clauses$ **where**
$learned\text{-}clss \equiv \lambda(\text{-},\ \text{-},\ U,\ \text{-}).\ U$

definition $backtrack\text{-}lvl :: 'v\ cdcl_W\text{-}mset \Rightarrow nat$ **where**
$backtrack\text{-}lvl \equiv \lambda(\text{-},\ \text{-},\ \text{-},\ k,\ \text{-}).\ k$

definition $conflicting :: 'v\ cdcl_W\text{-}mset \Rightarrow\ 'v\ clause\ option$ **where**
$conflicting \equiv \lambda(\text{-},\ \text{-},\ \text{-},\ \text{-},\ C).\ C$

definition $cons\text{-}trail :: ('v,\ 'v\ clause)\ ann\text{-}lit \Rightarrow\ 'v\ cdcl_W\text{-}mset \Rightarrow\ 'v\ cdcl_W\text{-}mset$ **where**
$cons\text{-}trail \equiv \lambda L\ (M,\ R).\ (L\ \#\ M,\ R)$

definition *tl-trail* **where**
$tl\text{-}trail \equiv \lambda(M,\ R).\ (tl\ M,\ R)$

definition *add-learned-cls* **where**
$add\text{-}learned\text{-}cls \equiv \lambda C\ (M,\ N,\ U,\ R).\ (M,\ N,\ \{\#C\#\} + U,\ R)$

definition *remove-cls* **where**
$remove\text{-}cls \equiv \lambda C\ (M,\ N,\ U,\ R).\ (M,\ removeAll\text{-}mset\ C\ N,\ removeAll\text{-}mset\ C\ U,\ R)$

definition *update-backtrack-lvl* **where**
$update\text{-}backtrack\text{-}lvl \equiv \lambda k\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$

definition *update-conflicting* **where**
update-conflicting $\equiv \lambda D$ $(M,\ N,\ U,\ k,\ \text{-}).$ $(M,\ N,\ U,\ k,\ D)$

definition *init-state* **where**
init-state $\equiv \lambda N.$ $([],\ N,\ \{\#\},\ 0,\ None)$

lemmas $cdcl_W$*-mset-state* = *trail-def cons-trail-def tl-trail-def add-learned-cls-def remove-cls-def update-backtrack-lvl-def update-conflicting-def init-clss-def learned-clss-def backtrack-lvl-def conflicting-def init-state-def*

interpretation $cdcl_W$*-mset*: $state_W$*-ops* **where**
trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
⟨*proof*⟩

interpretation $cdcl_W$*-mset*: $state_W$ **where**
trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
⟨*proof*⟩

interpretation $cdcl_W$*-mset*: *conflict-driven-clause-learning*$_W$ **where**
trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*

$\langle proof \rangle$

lemma $cdcl_W$*-mset-state-eq-eq*: $cdcl_W$*-mset.state-eq* = (*op* =)
$\langle proof \rangle$

notation $cdcl_W$*-mset.state-eq* (**infix** ∼*m* *49*)

7.2.2 Abstract Relation and Relation Theorems

This locales makes the lifting from the relation defined with multiset R and the version with an abstract state *R-abs*. We are lifting many different relations (each rule and the the strategy).

locale *relation-implied-relation-abs* =
fixes
R :: $'v$ $cdcl_W$*-mset* ⇒ $'v$ $cdcl_W$*-mset* ⇒ *bool* **and**
R-abs :: *'st* ⇒ *'st* ⇒ *bool* **and**
state :: *'st* ⇒ $'v$ $cdcl_W$*-mset* **and**
inv :: $'v$ $cdcl_W$*-mset* ⇒ *bool*
assumes
relation-compatible-state:
inv (*state S*) ⟹ *R-abs S T* ⟹ *R* (*state S*) (*state T*) **and**
relation-compatible-abs:
⋀*S S' T*. *inv S* ⟹ *S* ∼*m state S'* ⟹ *R S T* ⟹ ∃ *U*. *R-abs S' U* ∧ *T* ∼*m state U* **and**
relation-invariant:
⋀*S T*. *R S T* ⟹ *inv S* ⟹ *inv T* **and**
relation-abs-right-compatible:
⋀*S T U*. *inv* (*state S*) ⟹ *R-abs S T* ⟹ *state T* ∼*m state U* ⟹ *R-abs S U*
begin

lemma *relation-compatible-eq*:
assumes
inv: *inv* (*state S*) **and**
abs: *R-abs S T* **and**
SS': *state S* ∼*m state S'* **and**
TT': *state T* ∼*m state T'*
shows *R-abs S' T'*
$\langle proof \rangle$

lemma *rtranclp-relation-invariant*:
R^{++} *S T* ⟹ *inv S* ⟹ *inv T*
$\langle proof \rangle$

lemma *rtranclp-abs-rtranclp*:
$R\text{-}abs^{**}$ *S T* ⟹ *inv* (*state S*) ⟹ R^{**} (*state S*) (*state T*)
$\langle proof \rangle$

lemma *tranclp-relation-tranclp-relation-abs-compatible*:
fixes *S* :: *'st*
assumes
R: R^{++} (*state S*) *T* **and**
inv: *inv* (*state S*)
shows ∃ *U*. $R\text{-}abs^{++}$ *S U* ∧ *T* ∼*m state U*
$\langle proof \rangle$

lemma *rtranclp-relation-rtranclp-relation-abs-compatible*:
fixes *S* :: *'st*

assumes
  R: R^{**} $(state\ S)$ T **and**
  inv: inv $(state\ S)$
shows $\exists\, U.\ R\text{-}abs^{**}\ S\ U \wedge T \sim m\ state\ U$
$\langle proof \rangle$

lemma *no-step-iff*:
  $inv\ (state\ S) \Longrightarrow no\text{-}step\ R\ (state\ S) \longleftrightarrow no\text{-}step\ R\text{-}abs\ S$
  $\langle proof \rangle$

lemma *tranclp-relation-compatible-eq-and-inv*:
  assumes
    inv: $inv\ (state\ S)$ **and**
    st: $R\text{-}abs^{++}\ S\ T$ **and**
    SS': $state\ S \sim m\ state\ S'$ **and**
    TU: $state\ T \sim m\ state\ U$
  shows $R\text{-}abs^{++}\ S'\ U \wedge inv\ (state\ U)$
  $\langle proof \rangle$

lemma
  assumes
    inv: $inv\ (state\ S)$ **and**
    st: $R\text{-}abs^{++}\ S\ T$ **and**
    SS': $state\ S \sim m\ state\ S'$ **and**
    TU: $state\ T \sim m\ state\ U$
  shows
    tranclp-relation-compatible-eq: $R\text{-}abs^{++}\ S'\ U$ **and**
    tranclp-relation-abs-invariant: $inv\ (state\ U)$
    $\langle proof \rangle$

lemma *tranclp-abs-tranclp*: $R\text{-}abs^{++}\ S\ T \Longrightarrow inv\ (state\ S) \Longrightarrow R^{++}\ (state\ S)\ (state\ T)$
  $\langle proof \rangle$

lemma *full1-iff*:
  assumes inv: $inv\ (state\ S)$
  shows $full1\ R\ (state\ S)\ (state\ T) \longleftrightarrow full1\ R\text{-}abs\ S\ T$ (**is** $?R \longleftrightarrow ?R\text{-}abs$)
$\langle proof \rangle$

lemma *full1-iff-compatible*:
  assumes inv: $inv\ (state\ S)$ **and** SS': $S' \sim m\ state\ S$ **and** TT': $T' \sim m\ state\ T$
  shows $full1\ R\ S'\ T' \longleftrightarrow full1\ R\text{-}abs\ S\ T$ (**is** $?R \longleftrightarrow ?R\text{-}abs$)
  $\langle proof \rangle$

lemma *full-if-full-abs*:
  assumes $inv\ (state\ S)$ **and** $full\ R\text{-}abs\ S\ T$
  shows $full\ R\ (state\ S)\ (state\ T)$
  $\langle proof \rangle$

The converse does *not* hold, since we cannot prove that $S = T$ given $state\ S = state\ S$.

lemma *full-abs-if-full*:
  assumes $inv\ (state\ S)$ **and** $full\ R\ (state\ S)\ (state\ T)$
  shows $full\ R\text{-}abs\ S\ T \vee (state\ S \sim m\ state\ T \wedge no\text{-}step\ R\ (state\ S))$
  $\langle proof \rangle$

lemma *full-exists-full-abs*:
  assumes inv: $inv\ (state\ S)$ **and** $full$: $full\ R\ (state\ S)\ T$

obtains U **where** *full R-abs S U* **and** $T \sim m$ *state U*
⟨*proof*⟩

lemma *full1-exists-full1-abs*:
assumes *inv*: *inv* (*state S*) **and** *full1*: *full1 R* (*state S*) T
obtains U **where** *full1 R-abs S U* **and** $T \sim m$ *state U*
⟨*proof*⟩

lemma *full1-right-compatible*:
assumes *inv* (*state S*) **and**
full1: *full1 R-abs S T* **and** *TV*: *state T* $\sim m$ *state V*
shows *full1 R-abs S V*
⟨*proof*⟩

lemma *full-right-compatible*:
assumes *inv*: *inv* (*state S*) **and**
full-ST: *full R-abs S T* **and** *TU*: *state T* $\sim m$ *state U*
shows *full R-abs S U* $\vee$ ($S = T \wedge$ *no-step R-abs S*)
⟨*proof*⟩

end

locale *relation-relation-abs* =
fixes
$R :: 'v\ cdcl_W\text{-}mset \Rightarrow 'v\ cdcl_W\text{-}mset \Rightarrow bool$ **and**
$R\text{-}abs :: 'st \Rightarrow 'st \Rightarrow bool$ **and**
$state :: 'st \Rightarrow 'v\ cdcl_W\text{-}mset$ **and**
$inv :: 'v\ cdcl_W\text{-}mset \Rightarrow bool$
assumes
relation-compatible-state:
$inv\ (state\ S) \Longrightarrow R\ (state\ S)\ (state\ T) \longleftrightarrow R\text{-}abs\ S\ T$ **and**
relation-compatible-abs:
$\bigwedge S\ S'\ T.\ inv\ S \Longrightarrow S \sim m\ state\ S' \Longrightarrow R\ S\ T \Longrightarrow \exists\, U.\ R\text{-}abs\ S'\ U \wedge T \sim m\ state\ U$ **and**
relation-invariant:
$\bigwedge S\ T.\ R\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
begin

lemma *relation-compatible-eq*:
$inv\ (state\ S) \Longrightarrow R\text{-}abs\ S\ T \Longrightarrow state\ S \sim m\ state\ S' \Longrightarrow state\ T \sim m\ state\ T' \Longrightarrow R\text{-}abs\ S'\ T'$
⟨*proof*⟩

lemma *relation-right-compatible*:
$inv\ (state\ S) \Longrightarrow R\text{-}abs\ S\ T \Longrightarrow state\ T \sim m\ state\ U \Longrightarrow R\text{-}abs\ S\ U$
⟨*proof*⟩

sublocale *relation-implied-relation-abs*
⟨*proof*⟩

end

7.2.3 The State

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

locale *abs-state*$_W$*-clss-ops* =
  raw-clss get-lit mset-cls
    get-cls mset-clss
    +
  raw-cls mset-ccls
  for
    — Clause:
    get-lit :: $'cls \Rightarrow 'lit \Rightarrow 'v$ *literal option* **and**
    mset-cls :: $'cls \Rightarrow 'v$ *clause* **and**

    — Multiset of Clauses:
    get-cls :: $'clss \Rightarrow 'cls\text{-}it \Rightarrow 'cls$ *option* **and**
    mset-clss:: $'clss \Rightarrow 'cls$ *multiset* **and**

    — Conflicting clause:
    mset-ccls :: $'ccls \Rightarrow 'v$ *clause*
begin

fun *mmset-of-mlit* :: $'clss \Rightarrow ('v, 'cls\text{-}it)$ *ann-lit* $\Rightarrow ('v, 'v$ *clause*) *ann-lit*
  where
mmset-of-mlit Cs (*Propagated L C*) = *Propagated L* (*mset-cls* (*Cs* ⇓ *C*)) |
mmset-of-mlit - (*Decided L*) = *Decided L*

lemma *lit-of-mmset-of-mlit*[*simp*]:
  lit-of (*mmset-of-mlit Cs a*) = *lit-of a*
  ⟨*proof*⟩

lemma *lit-of-mmset-of-mlit-set-lit-of-l*[*simp*]:
  lit-of ‘ *mmset-of-mlit Cs* ‘ *set M'* = *lits-of-l M'*
  ⟨*proof*⟩

lemma *map-mmset-of-mlit-true-annots-true-cls*[*simp*]:
  map (*mmset-of-mlit Cs*) *M'* ⊨*as C* ⟷ *M'* ⊨*as C*
  ⟨*proof*⟩

definition *clauses-of-clss* **where**
clauses-of-clss N ≡ *image-mset mset-cls* (*mset-clss N*)

notation *cls-lit* (**infix** ↓ *49*)
notation *clss-cls* (**infix** ⇓ *49*)
notation *in-cls* (**infix** ∈↓ *49*)
notation *in-clss* (**infix** ∈⇓ *49*)
end

locale *abs-state*$_W$*-ops* =
  abs-state$_W$*-clss-ops*
    — functions for clauses:
    cls-lit mset-cls
    clss-cls mset-clss

    — functions for the conflicting clause:
    mset-ccls
  for
    — Clause:

cls-lit :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
mset-cls :: *'cls* ⇒ *'v clause* **and**

— Multiset of Clauses:
clss-cls :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
mset-clss:: *'clss* ⇒ *'cls multiset* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause* +
fixes
conc-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-conc-trail :: *'st* ⇒ (*'v*, *'cls-it*) *ann-lit* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
conc-backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conc-conflicting :: *'st* ⇒ *'ccls option* **and**

conc-learned-clss :: *'st* ⇒ *'v clauses* **and**

cons-conc-trail :: (*'v*, *'cls-it*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-conc-trail :: *'st* ⇒ *'st* **and**
add-conc-confl-to-learned-cls :: *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-conc-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
mark-conflicting :: *'cls-it* ⇒ *'st* ⇒ *'st* **and**
reduce-conc-trail-to :: (*'v*, *'v clause*) *ann-lits* ⇒ *'st* ⇒ *'st* **and**
resolve-conflicting :: *'v literal* ⇒ *'cls* ⇒ *'st* ⇒ *'st* **and**

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

definition *conc-clauses* :: *'st* ⇒ *'v clauses* **where**
conc-clauses S ≡ *image-mset mset-cls* (*mset-clss* (*raw-clauses S*))

definition *conc-init-clss* :: *'st* ⇒ *'v literal multiset multiset* **where**
conc-init-clss = (λ*S*. *conc-clauses S* − *conc-learned-clss S*)

abbreviation *conc-conflicting* :: *'st* ⇒ *'v clause option* **where**
conc-conflicting ≡ λ*S*. *map-option mset-ccls* (*raw-conc-conflicting S*)

definition *state* :: *'st* ⇒ *'v* $cdcl_W$*-mset* **where**
state = (λ*S*. (*conc-trail S*, *conc-init-clss S*, *conc-learned-clss S*, *conc-backtrack-lvl S*,
conc-conflicting S))

fun *valid-annotation* :: *'st* ⇒ (*'a*, *'cls-it*) *ann-lit* ⇒ *bool* **where**
valid-annotation S (*Propagated - E*) ⟷ *E* ∈⇓ (*raw-clauses S*) |
valid-annotation S (*Decided -*) ⟷ *True*

end

We are using an abstract state to abstract away the detail of the implementation: we do not need to know how the clauses are represented internally, we just need to know that they can be converted to multisets.

Weidenbach state is a five-tuple composed of:

1. the trail is a list of decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause (*'ccls*, standing for conflicting clause) and one for the initial and learned clauses (*'cls*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to *'v CDCL-Abstract-Clause-Representation.clause* is enough (needed for function *hd-raw-conc-trail* below).

There are several axioms to state the independance of the different fields of the state: for example, adding a clause to the learned clauses does not change the trail.

We define the following operations on the elements

- trail: *cons-trail*, *tl-trail*, and *reduce-conc-trail-to*.
- initial set of clauses: a clause can be removed.
- learned clauses: *add-conc-confl-to-learned-cls* moves the conflicting clause to the learned clauses.
- backtrack level: it can be arbitrary set.
- conflicting clause: there is *resolve-conflicting* that does a resolve step, *mark-conflicting* setting a conflict, and *add-conc-confl-to-learned-cls* setting the conflicting clause to *None*.

To ease the representation, we consider the clauses all together, where some of them are learned. This eases representation like arrays where the initial set of clause is at the beginning and avoid having an explicit *op* ∪ operator.

locale *abs-state*$_W$ =
abs-state$_W$*-ops*
— functions for clauses:
cls-lit mset-cls
clss-cls mset-clss

— functions for the conflicting clause:
mset-ccls

— functions about the state:
— getter:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss
— setter:
cons-conc-trail tl-conc-trail add-conc-confl-to-learned-cls remove-cls update-conc-backtrack-lvl
mark-conflicting reduce-conc-trail-to resolve-conflicting

— Some specific states:
conc-init-state

restart-state
for
— Clause:
$cls\text{-}lit :: 'cls \Rightarrow 'lit \Rightarrow 'v\ literal\ option$ **and**
$mset\text{-}cls :: 'cls \Rightarrow 'v\ clause$ **and**

— Multiset of Clauses:
$clss\text{-}cls :: 'clss \Rightarrow 'cls\text{-}it \Rightarrow 'cls\ option$ **and**
$mset\text{-}clss :: 'clss \Rightarrow 'cls\ multiset$ **and**

— Conflicting clause:
$mset\text{-}ccls :: 'ccls \Rightarrow 'v\ clause$ **and**

$conc\text{-}trail :: 'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lits$ **and**
$hd\text{-}raw\text{-}conc\text{-}trail :: 'st \Rightarrow ('v, 'cls\text{-}it)\ ann\text{-}lit$ **and**
$raw\text{-}clauses :: 'st \Rightarrow 'clss$ **and**
$conc\text{-}backtrack\text{-}lvl :: 'st \Rightarrow nat$ **and**
$raw\text{-}conc\text{-}conflicting :: 'st \Rightarrow 'ccls\ option$ **and**
$conc\text{-}learned\text{-}clss :: 'st \Rightarrow 'v\ clauses$ **and**

$cons\text{-}conc\text{-}trail :: ('v, 'cls\text{-}it)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$tl\text{-}conc\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}conc\text{-}confl\text{-}to\text{-}learned\text{-}cls :: 'st \Rightarrow 'st$ **and**
$remove\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}conc\text{-}backtrack\text{-}lvl :: nat \Rightarrow 'st \Rightarrow 'st$ **and**
$mark\text{-}conflicting :: 'cls\text{-}it \Rightarrow 'st \Rightarrow 'st$ **and**
$reduce\text{-}conc\text{-}trail\text{-}to :: ('v, 'v\ clause)\ ann\text{-}lits \Rightarrow 'st \Rightarrow 'st$ **and**
$resolve\text{-}conflicting :: 'v\ literal \Rightarrow 'cls \Rightarrow 'st \Rightarrow 'st$ **and**

$conc\text{-}init\text{-}state :: 'clss \Rightarrow 'st$ **and**
$restart\text{-}state :: 'st \Rightarrow 'st$ +
assumes
— Definition of *hd-raw-trail*:
hd-raw-conc-trail:
$conc\text{-}trail\ st \neq [] \Longrightarrow$
$mmset\text{-}of\text{-}mlit\ (raw\text{-}clauses\ st)\ (hd\text{-}raw\text{-}conc\text{-}trail\ st) = hd\ (conc\text{-}trail\ st)$ **and**

cons-conc-trail:
$\bigwedge S'.\ undefined\text{-}lit\ (conc\text{-}trail\ st)\ (lit\text{-}of\ L) \Longrightarrow$
$state\ st = (M, S') \Longrightarrow valid\text{-}annotation\ st\ L \Longrightarrow$
$state\ (cons\text{-}conc\text{-}trail\ L\ st) = (mmset\text{-}of\text{-}mlit\ (raw\text{-}clauses\ st)\ L\ \#\ M, S')$ **and**

tl-conc-trail:
$\bigwedge S'.\ state\ st = (M, S') \Longrightarrow state\ (tl\text{-}conc\text{-}trail\ st) = (tl\ M, S')$ **and**

remove-cls:
$\bigwedge S'.\ state\ st = (M, N, U, S') \Longrightarrow$
$state\ (remove\text{-}cls\ C\ st) =$
$(M, removeAll\text{-}mset\ (mset\text{-}cls\ C)\ N, removeAll\text{-}mset\ (mset\text{-}cls\ C)\ U, S')$ **and**

add-conc-confl-to-learned-cls:
$no\text{-}dup\ (conc\text{-}trail\ st) \Longrightarrow state\ st = (M, N, U, k, Some\ F) \Longrightarrow$
$state\ (add\text{-}conc\text{-}confl\text{-}to\text{-}learned\text{-}cls\ st) =$
$(M, N, \{\#F\#\} + U, k, None)$ **and**

update-conc-backtrack-lvl:

$\bigwedge S'$. *state st* = (*M*, *N*, *U*, *k*, *S'*) $\Longrightarrow$
state (*update-conc-backtrack-lvl k' st*) = (*M*, *N*, *U*, *k'*, *S'*) **and**

mark-conflicting:
state st = (*M*, *N*, *U*, *k*, *None*) $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
state (*mark-conflicting E st*) = (*M*, *N*, *U*, *k*, *Some* (*mset-cls* (*raw-clauses st* $\Downarrow$ *E*))) **and**

resolve-conflicting:
state st = (*M*, *N*, *U*, *k*, *Some F*) $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ *L'* $\in\#$ *mset-cls D* $\Longrightarrow$
state (*resolve-conflicting L' D st*) =
(*M*, *N*, *U*, *k*, *Some* ($cdcl_W$*-mset.resolve-cls L' F* (*mset-cls D*))) **and**

conc-init-state:
state (*conc-init-state Ns*) = ([], *clauses-of-clss Ns*, {#}, *0*, *None*) **and**

— Properties about restarting *restart-state*:
conc-trail-restart-state[*simp*]: *conc-trail* (*restart-state S*) = [] **and**
conc-init-clss-restart-state[*simp*]: *conc-init-clss* (*restart-state S*) = *conc-init-clss S* **and**
conc-learned-clss-restart-state[*intro*]:
conc-learned-clss (*restart-state S*) $\subseteq\#$ *conc-learned-clss S* **and**
conc-backtrack-lvl-restart-state[*simp*]: *conc-backtrack-lvl* (*restart-state S*) = *0* **and**
conc-conflicting-restart-state[*simp*]: *conc-conflicting* (*restart-state S*) = *None* **and**

— Properties about *reduce-conc-trail-to*:
reduce-conc-trail-to[*simp*]:
$\bigwedge S'$. *conc-trail st* = *M2* @ *M1* $\Longrightarrow$ *state st* = (*M*, *S'*) $\Longrightarrow$
state (*reduce-conc-trail-to M1 st*) = (*M1*, *S'*) **and**

learned-clauses:
conc-learned-clss S $\subseteq\#$ *conc-clauses S*
begin

lemma
conc-init-clss-tl-conc-trail[*simp*]:
conc-init-clss (*tl-conc-trail st*) = *conc-init-clss st* **and**
conc-init-clss-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* $\neq$ *None* $\Longrightarrow$
conc-init-clss (*add-conc-confl-to-learned-cls st*) = *conc-init-clss st* **and**
conc-init-clss-remove-cls[*simp*]:
conc-init-clss (*remove-cls C st*) = *removeAll-mset* (*mset-cls C*) (*conc-init-clss st*) **and**
conc-init-clss-update-conc-backtrack-lvl[*simp*]:
conc-init-clss (*update-conc-backtrack-lvl k st*) = *conc-init-clss st* **and**
conc-init-clss-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-init-clss (*mark-conflicting E st*) = *conc-init-clss st* **and**
conc-init-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ *L'* $\in\#$ *mset-cls D* $\Longrightarrow$
conc-init-clss (*resolve-conflicting L' D st*) = *conc-init-clss st* **and**
conc-init-clss-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-init-clss (*reduce-conc-trail-to M1 st*) = *conc-init-clss st*
⟨*proof*⟩

lemma
— Properties about the trail *conc-trail*:
conc-trail-cons-conc-trail[*simp*]:

undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-trail (*cons-conc-trail L st*) = *mmset-of-mlit* (*raw-clauses st*) *L* # *conc-trail st* **and**
conc-trail-tl-conc-trail[*simp*]:
conc-trail (*tl-conc-trail st*) = *tl* (*conc-trail st*) **and**
conc-trail-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* $\neq$ *None* $\Longrightarrow$
conc-trail (*add-conc-confl-to-learned-cls st*) = *conc-trail st* **and**
conc-trail-remove-cls[*simp*]:
conc-trail (*remove-cls C st*) = *conc-trail st* **and**
conc-trail-update-conc-backtrack-lvl[*simp*]:
conc-trail (*update-conc-backtrack-lvl k st*) = *conc-trail st* **and**
conc-trail-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-trail (*mark-conflicting E st*) = *conc-trail st* **and**
conc-trail-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L' \in\#$ *F* $\Longrightarrow$ $L' \in\#$ *mset-cls D* $\Longrightarrow$
conc-trail (*resolve-conflicting L' D st*) = *conc-trail st* **and**

— Properties about the initial clauses *conc-init-clss*:
conc-init-clss-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-init-clss (*cons-conc-trail L st*) = *conc-init-clss st*
and

— Properties about the learned clauses *conc-learned-clss*:
conc-learned-clss-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-learned-clss (*cons-conc-trail L st*) = *conc-learned-clss st* **and**
conc-learned-clss-tl-conc-trail[*simp*]:
conc-learned-clss (*tl-conc-trail st*) = *conc-learned-clss st* **and**
conc-learned-clss-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* = *Some C'* $\Longrightarrow$
conc-learned-clss (*add-conc-confl-to-learned-cls st*) = $\{\#C'\#\}$ + *conc-learned-clss st* **and**
conc-learned-clss-remove-cls[*simp*]:
conc-learned-clss (*remove-cls C st*) = *removeAll-mset* (*mset-cls C*) (*conc-learned-clss st*) **and**
conc-learned-clss-update-conc-backtrack-lvl[*simp*]:
conc-learned-clss (*update-conc-backtrack-lvl k st*) = *conc-learned-clss st* **and**
conc-learned-clss-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-learned-clss (*mark-conflicting E st*) = *conc-learned-clss st* **and**
conc-learned-clss-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L' \in\#$ *F* $\Longrightarrow$ $L' \in\#$ *mset-cls D* $\Longrightarrow$
conc-learned-clss (*resolve-conflicting L' D st*) = *conc-learned-clss st* **and**

— Properties about the backtracking level *conc-backtrack-lvl*:
conc-backtrack-lvl-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-backtrack-lvl (*cons-conc-trail L st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-tl-conc-trail[*simp*]:
conc-backtrack-lvl (*tl-conc-trail st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* $\neq$ *None* $\Longrightarrow$
conc-backtrack-lvl (*add-conc-confl-to-learned-cls st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-remove-cls[*simp*]:
conc-backtrack-lvl (*remove-cls C st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-update-conc-backtrack-lvl[*simp*]:

conc-backtrack-lvl (*update-conc-backtrack-lvl k st*) = *k* **and**
conc-backtrack-lvl-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-backtrack-lvl (*mark-conflicting E st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-clss-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ L' $\in\#$ *mset-cls D* $\Longrightarrow$
conc-backtrack-lvl (*resolve-conflicting* L' *D st*) = *conc-backtrack-lvl st* **and**

— Properties about the conflicting clause *conc-conflicting*:
conc-conflicting-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-conflicting (*cons-conc-trail L st*) = *conc-conflicting st* **and**
conc-conflicting-tl-conc-trail[*simp*]:
conc-conflicting (*tl-conc-trail st*) = *conc-conflicting st* **and**
conc-conflicting-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* = *Some* C' $\Longrightarrow$
conc-conflicting (*add-conc-confl-to-learned-cls st*) = *None*
and
raw-conc-conflicting-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* = *Some* C' $\Longrightarrow$
raw-conc-conflicting (*add-conc-confl-to-learned-cls st*) = *None* **and**
conc-conflicting-remove-cls[*simp*]:
conc-conflicting (*remove-cls C st*) = *conc-conflicting st* **and**
conc-conflicting-update-conc-backtrack-lvl[*simp*]:
conc-conflicting (*update-conc-backtrack-lvl k st*) = *conc-conflicting st* **and**
conc-conflicting-clss-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ L' $\in\#$ *mset-cls D* $\Longrightarrow$
conc-conflicting (*resolve-conflicting* L' *D st*) =
Some ($cdcl_W$*-mset.resolve-cls* L' *F* (*mset-cls D*)) **and**

— Properties about the initial state *conc-init-state*:
conc-init-state-conc-trail[*simp*]: *conc-trail* (*conc-init-state Ns*) = [] **and**
conc-init-state-clss[*simp*]: *conc-init-clss* (*conc-init-state Ns*) = *clauses-of-clss Ns* **and**
conc-init-state-conc-learned-clss[*simp*]: *conc-learned-clss* (*conc-init-state Ns*) = {#} **and**
conc-init-state-conc-backtrack-lvl[*simp*]: *conc-backtrack-lvl* (*conc-init-state Ns*) = *0* **and**
conc-init-state-conc-conflicting[*simp*]: *conc-conflicting* (*conc-init-state Ns*) = *None* **and**

— Properties about *reduce-conc-trail-to*:
trail-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$ *conc-trail* (*reduce-conc-trail-to M1 st*) = *M1* **and**
conc-learned-clss-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-learned-clss (*reduce-conc-trail-to M1 st*) = *conc-learned-clss st* **and**
conc-backtrack-lvl-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-backtrack-lvl (*reduce-conc-trail-to M1 st*) = *conc-backtrack-lvl st* **and**
conc-conflicting-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-conflicting (*reduce-conc-trail-to M1 st*) = *conc-conflicting st*
⟨*proof*⟩

abbreviation *incr-lvl* :: *'st* $\Rightarrow$ *'st* **where**
incr-lvl S $\equiv$ *update-conc-backtrack-lvl* (*conc-backtrack-lvl S* + *1*) *S*

abbreviation *state-eq* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* (**infix** $\sim$ *36*) **where**

$S \sim T \equiv$ *state* $S \sim m$ *state* T

lemma *state-eq-sym*:
$S \sim T \longleftrightarrow T \sim S$
⟨*proof*⟩

lemma *state-eq-trans*:
$S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$
⟨*proof*⟩

lemma *conc-clauses-init-learned*: *conc-clauses* S = *conc-init-clss* S + *conc-learned-clss* S
⟨*proof*⟩

lemma
init-clss-conc-init-clss[*simp*]:
init-clss (*state* S) = *conc-init-clss* S **and**
learned-clss-conc-learned-clss[*simp*]:
learned-clss (*state* S) = *conc-learned-clss* S
⟨*proof*⟩

lemma *clauses-conc-clauses*[*simp*]:
$cdcl_W$*-mset.clauses* (*state* S) = *conc-clauses* S
⟨*proof*⟩

lemma
backtrack-lvl-conc-backtrack-lvl[*simp*]:
backtrack-lvl (*state* S) = *conc-backtrack-lvl* S **and**
trail-conc-trail[*simp*]:
trail (*state* S) = *conc-trail* S **and**
conflicting-conc-conflicting[*simp*]:
conflicting (*state* S) = *conc-conflicting* S
⟨*proof*⟩

lemma
shows
state-eq-conc-trail: $S \sim T \Longrightarrow$ *conc-trail* S = *conc-trail* T **and**
state-eq-conc-init-clss: $S \sim T \Longrightarrow$ *conc-init-clss* S = *conc-init-clss* T **and**
state-eq-conc-learned-clss: $S \sim T \Longrightarrow$ *conc-learned-clss* S = *conc-learned-clss* T **and**
state-eq-conc-backtrack-lvl: $S \sim T \Longrightarrow$ *conc-backtrack-lvl* S = *conc-backtrack-lvl* T **and**
state-eq-conc-conflicting: $S \sim T \Longrightarrow$ *conc-conflicting* S = *conc-conflicting* T **and**
state-eq-clauses: $S \sim T \Longrightarrow$ *conc-clauses* S = *conc-clauses* T **and**
state-eq-undefined-lit:
$S \sim T \Longrightarrow$ *undefined-lit* (*conc-trail* S) L = *undefined-lit* (*conc-trail* T) L
⟨*proof*⟩

We combine all simplification rules about *op* $\sim$ in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

lemmas *state-simp* = *state-eq-conc-trail state-eq-conc-init-clss state-eq-conc-learned-clss state-eq-conc-backtrack-lvl state-eq-conc-conflicting state-eq-clauses state-eq-undefined-lit*

lemma *atms-of-ms-conc-learned-clss-restart-state-in-atms-of-ms-conc-learned-clssI*[*intro*]:
$x \in$ *atms-of-mm* (*conc-learned-clss* (*restart-state* S)) $\Longrightarrow x \in$ *atms-of-mm* (*conc-learned-clss* S)
⟨*proof*⟩

lemma *clauses-reduce-conc-trail-to*[*simp*]:
conc-trail S = *M2* @ *M1* ⟹ *conc-clauses* (*reduce-conc-trail-to M1 S*) = *conc-clauses S*
⟨*proof*⟩

lemma *in-get-all-ann-decomposition-clauses-reduce-conc-trail-to*[*simp*]:
(*L* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail S*)) ⟹
conc-clauses (*reduce-conc-trail-to M1 S*) = *conc-clauses S*
⟨*proof*⟩

lemma *in-get-all-ann-decomposition-conc-trail-update-conc-trail*[*simp*]:
assumes *H*: (*L* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail S*))
shows *conc-trail* (*reduce-conc-trail-to M1 S*) = *M1*
⟨*proof*⟩

lemma *raw-conc-conflicting-cons-conc-trail*[*simp*]:
assumes *undefined-lit* (*conc-trail S*) (*lit-of L*) **and** *valid-annotation S L*
shows
raw-conc-conflicting (*cons-conc-trail L S*) = *None* ⟷ *raw-conc-conflicting S* = *None*
⟨*proof*⟩

lemma *raw-conc-conflicting-update-backtracl-lvl*[*simp*]:
raw-conc-conflicting (*update-conc-backtrack-lvl k S*) = *None* ⟷ *raw-conc-conflicting S* = *None*
⟨*proof*⟩

lemma *conc-conflicting-mark-conflicting*[*simp*]:
raw-conc-conflicting S = *None* ⟹ *E* ∈⇓ *raw-clauses S* ⟹
conc-conflicting (*mark-conflicting E S*) = *Some* (*mset-cls* (*raw-clauses S* ⇓ *E*))
⟨*proof*⟩

lemma *conflicting-None-iff-raw-conc-conflicting*[*simp*]:
conflicting (*state S*) = *None* ⟷ *raw-conc-conflicting S* = *None*
⟨*proof*⟩

lemma *trail-state-add-conc-confl-to-learned-cls*:
no-dup (*conc-trail S*) ⟹ *conc-conflicting S* ≠ *None* ⟹
trail (*state* (*add-conc-confl-to-learned-cls S*)) = *trail* (*state S*)
⟨*proof*⟩

lemma *trail-state-update-backtrack-lvl*:
trail (*state* (*update-conc-backtrack-lvl i S*)) = *trail* (*state S*)
⟨*proof*⟩

lemma *trail-state-update-conflicting*:
raw-conc-conflicting S = *None* ⟹ *E* ∈⇓ *raw-clauses S* ⟹
trail (*state* (*mark-conflicting E S*)) = *trail* (*state S*)
⟨*proof*⟩

lemma *tl-trail-state-tl-con-trail*[*simp*]:
tl-trail (*state S*) = *state* (*tl-conc-trail S*)
⟨*proof*⟩

lemma *add-learned-cls-state-add-conc-confl-to-learned-cls*[*simp*]:
assumes *no-dup* (*conc-trail S*) **and** *raw-conc-conflicting S* = *Some D*
shows *update-conflicting None* (*add-learned-cls* (*mset-ccls D*) (*state S*)) =
state (*add-conc-confl-to-learned-cls S*)
⟨*proof*⟩

lemma *state-cons-cons-trail-cons-trail*[*simp*]:
undefined-lit (*trail* (*state S*)) (*lit-of L*) $\Longrightarrow$ *valid-annotation S L* $\Longrightarrow$
cons-trail (*mmset-of-mlit* (*raw-clauses S*) *L*) (*state S*) = *state* (*cons-conc-trail L S*)
⟨*proof*⟩

lemma *state-cons-cons-trail-cons-trail-propagated*[*simp*]:
undefined-lit (*trail* (*state S*)) *K* $\Longrightarrow$ *C* $\in\Downarrow$ *raw-clauses S* $\Longrightarrow$
cons-trail (*Propagated K* (*mset-cls* (*raw-clauses S* $\Downarrow$ *C*))) (*state S*)
= *state* (*cons-conc-trail* (*Propagated K C*) *S*)
⟨*proof*⟩

lemma *state-cons-cons-trail-cons-trail-decided*[*simp*]:
undefined-lit (*trail* (*state S*)) *K* $\Longrightarrow$
cons-trail (*Decided K*) (*state S*) = *state* (*cons-conc-trail* (*Decided K*) *S*)
⟨*proof*⟩

lemma *state-mark-conflicting-update-conflicting*[*simp*]:
assumes *raw-conc-conflicting S* = *None* **and** *D* $\in\Downarrow$ *raw-clauses S*
shows
update-conflicting (*Some* (*mset-cls* (*raw-clauses S* $\Downarrow$ *D*))) (*state S*) =
state (*mark-conflicting* (*D*) *S*)
⟨*proof*⟩

lemma *update-backtrack-lvl-state*[*simp*]:
update-backtrack-lvl i (*state S*) = *state* (*update-conc-backtrack-lvl i S*)
⟨*proof*⟩

lemma *update-conflicting-resolve-state-mark-conflicting*[*simp*]:
raw-conc-conflicting S = *Some D'* $\Longrightarrow$ $-L$ $\in\#$ *mset-ccls D'* $\Longrightarrow$ *L* $\in\#$ *mset-cls E'* $\Longrightarrow$
update-conflicting (*Some* (*remove1-mset* ($-$ *L*) (*mset-ccls D'*) $\#\cup$ *remove1-mset L* (*mset-cls E'*)))
(*state* (*tl-conc-trail S*)) =
state (*resolve-conflicting L E'* (*tl-conc-trail S*))
⟨*proof*⟩

lemma *add-learned-update-backtrack-update-conflicting*[*simp*]:
no-dup (*conc-trail S*) $\Longrightarrow$ *raw-conc-conflicting S* = *Some D* $\Longrightarrow$ *D'* $\in\Downarrow$ *T* $\Longrightarrow$
mset-cls (*T* $\Downarrow$ *D'*) = *mset-ccls D* $\Longrightarrow$
add-learned-cls (*mset-cls* (*T* $\Downarrow$ *D'*))
(*update-backtrack-lvl i*
(*update-conflicting None*
(*state S*))) =
state (*add-conc-confl-to-learned-cls* (*update-conc-backtrack-lvl i S*))
⟨*proof*⟩

lemma *state-state*:
$cdcl_W$*-mset.state* (*state S*) = (*trail* (*state S*), *init-clss* (*state S*), *learned-clss* (*state S*),
backtrack-lvl (*state S*), *conflicting* (*state S*))
⟨*proof*⟩

lemma *state-reduce-conc-trail-to-reduce-conc-trail-to*[*simp*]:
assumes [*simp*]: *conc-trail S* = *M2* @ *M1*
shows $cdcl_W$*-mset.reduce-trail-to M1* (*state S*) = *state* (*reduce-conc-trail-to M1 S*) (**is** *?RS* = *?SR*)
⟨*proof*⟩

lemma *state-conc-init-state*: *state* (*conc-init-state N*) = *init-state* (*clauses-of-clss N*)

⟨*proof*⟩

lemma *conc-clauses-add-conc-confl-to-learned-cls*[*simp*]:
conc-conflicting S = *Some C* ⟹ *no-dup* (*conc-trail S*) ⟹
conc-clauses (*add-conc-confl-to-learned-cls S*) = {#*C*#} + *conc-clauses S*
⟨*proof*⟩

lemma *raw-conc-conflicting-update-conc-backtrack-lvl*:
raw-conc-conflicting (*update-conc-backtrack-lvl i S*) = *Some z'* ⟹
(*raw-conc-conflicting S* ≠ *None* ∧ *conc-conflicting S* = *Some* (*mset-ccls z'*))

⟨*proof*⟩

More robust version of theorem *in-mset-clss-exists-preimage*:

lemma *in-clauses-preimage*:
assumes *b*: *b* ∈# $cdcl_W$*-mset.clauses* (*state C*)
shows ∃ *b'*. *b'* ∈⇓ *raw-clauses C* ∧ *mset-cls* ((*raw-clauses C*) ⇓ *b'*) = *b*
⟨*proof*⟩

lemma *state-reduce-conc-trail-to-reduce-conc-trail-to-decomp*[*simp*]:
assumes (*P* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail S*))
shows $cdcl_W$*-mset.reduce-trail-to M1* (*state S*) = *state* (*reduce-conc-trail-to M1 S*)
⟨*proof*⟩

end — end of *abs-state*$_W$ locale

7.2.4 CDCL Rules

locale *abs-conflict-driven-clause-learning*$_W$ =
abs-state$_W$
— functions for clauses:
get-lit mset-cls
get-cls mset-clss

— functions for the conflicting clause:
mset-ccls

— functions about the state:
— getter:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss
— setter:
cons-conc-trail tl-conc-trail add-conc-confl-to-learned-cls remove-cls update-conc-backtrack-lvl
mark-conflicting reduce-conc-trail-to resolve-conflicting

— Some specific states:
conc-init-state
restart-state
for
— Clause:
get-lit :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
mset-cls :: *'cls* ⇒ *'v clause* **and**

— Multiset of Clauses:
get-cls :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
mset-clss:: *'clss* ⇒ *'cls multiset* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause* **and**

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

cons-conc-trail :: (*'v*, *'cls-it*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-conc-trail :: *'st* ⇒ *'st* **and**
add-conc-confl-to-learned-cls :: *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-conc-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
mark-conflicting :: *'cls-it* ⇒ *'st* ⇒ *'st* **and**
reduce-conc-trail-to :: (*'v*, *'v clause*) *ann-lits* ⇒ *'st* ⇒ *'st* **and**
resolve-conflicting :: *'v literal* ⇒ *'cls* ⇒ *'st* ⇒ *'st* **and**

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

inductive *propagate-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate-abs-rule: *propagate-abs S T*
if
conc-conflicting S = *None* **and**
E ∈⇓ *raw-clauses S* **and**
L ∈# *mset-cls* (*raw-clauses S* ⇓ *E*) **and**
conc-trail S ⊨*as CNot* (*mset-cls* (*raw-clauses S* ⇓ *E*) − {#*L*#}) **and**
undefined-lit (*conc-trail S*) *L* **and**
T ∼ *cons-conc-trail* (*Propagated L E*) *S*

inductive-cases *propagate-absE*: *propagate-abs S T*

lemma *in-clss-mset-clss*:
assumes *H*: *a* ∈⇓ *Cs*
shows (*Cs* ⇓ *a*) ∈# *mset-clss Cs*
⟨*proof*⟩

lemma *propagate-propagate-abs*:
$cdcl_W$*-mset.propagate* (*state S*) (*state T*) ⟷ *propagate-abs S T* (**is** *?mset* ⟷ *?abs*)
⟨*proof*⟩

lemma *propagate-compatible-abs*:
assumes *SS'*: *S* ∼*m state S'* **and** *abs*: $cdcl_W$*-mset.propagate S T*
obtains *U* **where** *propagate-abs S' U* **and** *T* ∼*m state U*
⟨*proof*⟩

interpretation *propagate-abs*: *relation-relation-abs* $cdcl_W$*-mset.propagate propagate-abs state*
λ-. *True*
⟨*proof*⟩

inductive *conflict-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
conflict-abs-rule:

conc-conflicting S = None $\Longrightarrow$
D $\in\Downarrow$ *raw-clauses S* $\Longrightarrow$
conc-trail S $\models$*as CNot* (*mset-cls* (*raw-clauses S* $\Downarrow$ *D*)) $\Longrightarrow$
T $\sim$ *mark-conflicting D S* $\Longrightarrow$
conflict-abs S T

inductive-cases *conflict-absE*: *conflict-abs S T*

lemma *conflict-conflict-abs*:
cdcl$_W$*-mset.conflict* (*state S*) (*state T*) $\longleftrightarrow$ *conflict-abs S T* (**is** *?mset* $\longleftrightarrow$ *?abs*)
⟨*proof*⟩

lemma *conflict-compatible-abs*:
assumes *SS'*: *S* $\sim$*m state S'* **and** *conflict*: *cdcl*$_W$*-mset.conflict S T*
obtains *U* **where** *conflict-abs S' U* **and** *T* $\sim$*m state U*
⟨*proof*⟩

interpretation *conflict-abs*: *relation-relation-abs cdcl*$_W$*-mset.conflict conflict-abs state*
λ-. *True*
⟨*proof*⟩

In the backtrack rule, we assume the existence of an index D' such that the clause is equal to the one use to backtrack.

1. the clause D was added to the state by *add-conc-confl-to-learned-cls*
2. therefore, the index D' exists.

inductive *backtrack-abs* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
backtrack-abs-rule:
conc-conflicting S = Some D $\Longrightarrow$
L $\in$# *D* $\Longrightarrow$
(*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*conc-trail S*)) $\Longrightarrow$
get-level (*conc-trail S*) *L = conc-backtrack-lvl S* $\Longrightarrow$
get-level (*conc-trail S*) *L = get-maximum-level* (*conc-trail S*) *D* $\Longrightarrow$
get-maximum-level (*conc-trail S*) (*D* − {#*L*#}) $\equiv$ *i* $\Longrightarrow$
get-level (*conc-trail S*) *K = i + 1* $\Longrightarrow$
mset-cls
(*raw-clauses* (*reduce-conc-trail-to M1* (*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S*))) $\Downarrow$ *D'*) = *D* $\Longrightarrow$
D' $\in\Downarrow$ *raw-clauses* (*reduce-conc-trail-to M1* (*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S*))) $\Longrightarrow$
T $\sim$ *cons-conc-trail* (*Propagated L D'*)
(*reduce-conc-trail-to M1*
(*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S*))) $\Longrightarrow$
backtrack-abs S T

inductive-cases *backtrack-absE*: *backtrack-abs S T*

lemma *backtrack-backtrack-abs*:
assumes *inv*: *cdcl*$_W$*-mset.cdcl*$_W$*-all-struct-inv* (*state S*)
shows *cdcl*$_W$*-mset.backtrack* (*state S*) (*state T*) $\longleftrightarrow$ *backtrack-abs S T* (**is** *?conc* $\longleftrightarrow$ *?abs*)
⟨*proof*⟩

lemma *backtrack-exists-backtrack-abs-step*:

assumes *bt*: $cdcl_W$*-mset.backtrack* S T **and** *inv*: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* S **and**
SS': $S \sim m$ *state* S'
obtains U **where** *backtrack-abs* S' U **and** $T \sim m$ *state* U
⟨*proof*⟩

interpretation *backtrack-abs*: *relation-relation-abs* $cdcl_W$*-mset.backtrack backtrack-abs state*
$cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv*
⟨*proof*⟩

inductive *decide-abs* :: $'st \Rightarrow 'st \Rightarrow bool$ **for** S :: $'st$ **where**
decide-abs-rule:
conc-conflicting S = *None* $\Longrightarrow$
undefined-lit (*conc-trail* S) L $\Longrightarrow$
atm-of $L \in$ *atms-of-mm* (*conc-init-clss* S) $\Longrightarrow$
$T \sim$ *cons-conc-trail* (*Decided* L) (*incr-lvl* S) $\Longrightarrow$
decide-abs S T

inductive-cases *decide-absE*: *decide-abs* S T

lemma *decide-decide-abs*:
$cdcl_W$*-mset.decide* (*state* S) (*state* T) $\longleftrightarrow$ *decide-abs* S T
⟨*proof*⟩

interpretation *decide-abs*: *relation-relation-abs* $cdcl_W$*-mset.decide decide-abs state*
λ-. *True*
⟨*proof*⟩

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

inductive-cases *skip-absE*: *skip-abs* S T

lemma *skip-skip-abs*:
$cdcl_W$*-mset.skip* (*state* S) (*state* T) $\longleftrightarrow$ *skip-abs* S T (**is** *?conc* $\longleftrightarrow$ *?abs*)
⟨*proof*⟩

lemma *skip-exists-skip-abs*:
assumes *skip*: $cdcl_W$*-mset.skip* S T **and** *SS'*: $S \sim m$ *state* S'
obtains U **where** *skip-abs* S' U **and** $T \sim m$ *state* U
⟨*proof*⟩

interpretation *skip-abs*: *relation-relation-abs* $cdcl_W$*-mset.skip skip-abs state*
λ-. *True*
⟨*proof*⟩

inductive *resolve-abs* :: $'st \Rightarrow 'st \Rightarrow bool$ **for** S :: $'st$ **where**
resolve-abs-rule: *conc-trail* $S \neq []$ $\Longrightarrow$
hd-raw-conc-trail S = *Propagated* L E $\Longrightarrow$
$L \in\#$ *mset-cls* (*raw-clauses* $S \Downarrow E$) $\Longrightarrow$
raw-conc-conflicting S = *Some* D' $\Longrightarrow$

$-L \in\#$ *mset-ccls* $D' \Longrightarrow$
get-maximum-level (*conc-trail* S) (*remove1-mset* $(-L)$ (*mset-ccls* D')) = *conc-backtrack-lvl* $S \Longrightarrow$
$T \sim$ *resolve-conflicting* L (*raw-clauses* $S \Downarrow E$) (*tl-conc-trail* S) $\Longrightarrow$
resolve-abs S T

inductive-cases *resolve-absE*: *resolve-abs* S T

lemma *resolve-resolve-abs*:
$cdcl_W$*-mset.resolve* (*state* S) (*state* T) $\longleftrightarrow$ *resolve-abs* S T (**is** *?conc* $\longleftrightarrow$ *?abs*)
⟨*proof*⟩

lemma *resolve-exists-resolve-abs*:
assumes
res: $cdcl_W$*-mset.resolve* S T **and**
SS': $S \sim m$ *state* S'
obtains U **where** *resolve-abs* S' U **and** $T \sim m$ *state* U
⟨*proof*⟩

interpretation *resolve-abs*: *relation-relation-abs* $cdcl_W$*-mset.resolve* *resolve-abs* *state*
λ-. *True*
⟨*proof*⟩

inductive *restart* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
restart: *conc-conflicting* S = *None* $\Longrightarrow$
¬*conc-trail* S ⊨*asm* *conc-clauses* $S \Longrightarrow$
$T \sim$ *restart-state* $S \Longrightarrow$
restart S T

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

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

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

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

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

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

inductive-cases $cdcl_W$*-abs-bjE*: $cdcl_W$*-abs-bj* S T

lemma $cdcl_W$*-abs-bj-*$cdcl_W$*-abs-bj*:
$cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state* S) $\Longrightarrow$

$cdcl_W$*-mset.*$cdcl_W$*-bj* (*state S*) (*state T*) $\longleftrightarrow$ $cdcl_W$*-abs-bj S T*
⟨*proof*⟩

interpretation $cdcl_W$*-abs-bj*: *relation-relation-abs* $cdcl_W$*-mset.*$cdcl_W$*-bj* $cdcl_W$*-abs-bj state* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv*
⟨*proof*⟩

inductive $cdcl_W$*-abs-o* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
decide: *decide-abs S S'* ⟹ $cdcl_W$*-abs-o S S'* |
bj: $cdcl_W$*-abs-bj S S'* ⟹ $cdcl_W$*-abs-o S S'*

inductive $cdcl_W$*-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate: *propagate-abs S S'* ⟹ $cdcl_W$*-abs S S'* |
conflict: *conflict-abs S S'* ⟹ $cdcl_W$*-abs S S'* |
other: $cdcl_W$*-abs-o S S'* ⟹ $cdcl_W$*-abs S S'*|
rf: $cdcl_W$*-abs-rf S S'* ⟹ $cdcl_W$*-abs S S'*

7.2.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have add a strategy and show the inclusion in the multiset version.

inductive $cdcl_W$*-merge-abs-cp* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
conflict': *conflict-abs S T* ⟹ *full* $cdcl_W$*-abs-bj T U* ⟹ $cdcl_W$*-merge-abs-cp S U* |
propagate': *propagate-abs*$^{++}$ *S S'* ⟹ $cdcl_W$*-merge-abs-cp S S'*

lemma $cdcl_W$*-merge-cp-*$cdcl_W$*-abs-merge-cp*:
assumes
cp: $cdcl_W$*-merge-abs-cp S T* **and**
inv: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*)
shows $cdcl_W$*-mset.*$cdcl_W$*-merge-cp* (*state S*) (*state T*)
⟨*proof*⟩

lemma $cdcl_W$*-merge-cp-abs-exists-*$cdcl_W$*-merge-cp*:
assumes
cp: $cdcl_W$*-mset.*$cdcl_W$*-merge-cp* (*state S*) *T* **and**
inv: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*)
obtains *U* **where** $cdcl_W$*-merge-abs-cp S U* **and** *T* ∼*m state U*
⟨*proof*⟩

lemma *no-step-*$cdcl_W$*-merge-cp-no-step-*$cdcl_W$*-abs-merge-cp*:
assumes
inv: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*)
shows *no-step* $cdcl_W$*-merge-abs-cp S* $\longleftrightarrow$ *no-step* $cdcl_W$*-mset.*$cdcl_W$*-merge-cp* (*state S*)
(**is** *?abs* $\longleftrightarrow$ *?conc*)
⟨*proof*⟩

lemma $cdcl_W$*-merge-abs-cp-right-compatible*:
$cdcl_W$*-merge-abs-cp S V* ⟹ $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*) ⟹
V ∼ *W* ⟹ $cdcl_W$*-merge-abs-cp S W*
⟨*proof*⟩

interpretation $cdcl_W$*-merge-abs-cp*: *relation-implied-relation-abs*
$cdcl_W$*-mset.*$cdcl_W$*-merge-cp* $cdcl_W$*-merge-abs-cp state* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv*
⟨*proof*⟩

inductive *cdcl$_W$-merge-abs-stgy* **for** S :: *'st* **where**
fw-s-cp: *full1 cdcl$_W$-merge-abs-cp S T* $\Longrightarrow$ *cdcl$_W$-merge-abs-stgy S T* |
fw-s-decide: *decide-abs S T* $\Longrightarrow$ *no-step cdcl$_W$-merge-abs-cp S* $\Longrightarrow$ *full cdcl$_W$-merge-abs-cp T U*
$\Longrightarrow$ *cdcl$_W$-merge-abs-stgy S U*

lemma *cdcl$_W$-cp-cdcl$_W$-abs-cp*:
assumes *stgy*: *cdcl$_W$-merge-abs-stgy S T* **and**
inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*)
shows *cdcl$_W$-mset.cdcl$_W$-merge-stgy* (*state S*) (*state T*)
⟨*proof*⟩

lemma *cdcl$_W$-merge-abs-stgy-exists-cdcl$_W$-merge-stgy*:
assumes
inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv S* **and**
SS': $S \sim m$ *state* S' **and**
st: *cdcl$_W$-mset.cdcl$_W$-merge-stgy S T*
shows $\exists$ *U*. *cdcl$_W$-merge-abs-stgy S' U* $\wedge$ $T \sim m$ *state U*
⟨*proof*⟩

lemma *cdcl$_W$-merge-abs-stgy-right-compatible*:
assumes
inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*) **and**
st: *cdcl$_W$-merge-abs-stgy S T* **and**
TU: $T \sim V$
shows *cdcl$_W$-merge-abs-stgy S V*
⟨*proof*⟩

interpretation *cdcl$_W$-merge-abs-stgy*: *relation-implied-relation-abs*
cdcl$_W$-mset.cdcl$_W$-merge-stgy cdcl$_W$-merge-abs-stgy state cdcl$_W$-mset.cdcl$_W$-all-struct-inv
⟨*proof*⟩

lemma *cdcl$_W$-merge-abs-stgy-final-State-conclusive*:
fixes T :: *'st*
assumes
full: *full cdcl$_W$-merge-abs-stgy* (*conc-init-state N*) *T* **and**
n-d: *distinct-mset-mset* (*clauses-of-clss N*)
shows (*conc-conflicting T* = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*clauses-of-clss N*)))
$\vee$ (*conc-conflicting T* = *None* $\wedge$ *conc-trail T* $\models$*asm clauses-of-clss N*
$\wedge$ *satisfiable* (*set-mset* (*clauses-of-clss N*)))
⟨*proof*⟩

end

end

7.3 2-Watched-Literal

theory *CDCL-Two-Watched-Literals*
imports *CDCL-W-Abstract-State*
begin

Two-watched literals

datatype *'v twl-clause* =

$TWL\text{-}Clause$ (*watched*: $'v$) (*unwatched*: $'v$)

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* $\Rightarrow$ *bool* **where**
struct-wf-twl-cls ($TWL\text{-}Clause$ W UW) $\longleftrightarrow$
size $W \leq 2 \wedge$ (*size* $W < 2 \longrightarrow UW \subseteq\# W$) $\wedge$ *distinct-mset* ($W + UW$)

fun *clause* :: $'a$ *twl-clause* $\Rightarrow$ $'a$ :: {*plus*} **where**
clause ($TWL\text{-}Clause$ W UW) $= W + UW$

primrec (*nonexhaustive*) *index* :: $'a$ *list* $\Rightarrow 'a \Rightarrow$ *nat* **where**
index (a # l) c = (*if* $a = c$ *then* 0 *else* 1+*index* l c)

lemma *index-nth*:
$a \in$ *set* $l \Longrightarrow l$! (*index* l a) $= a$
⟨*proof*⟩

definition *defined-before* :: ($'a$, $'b$) *ann-lit list* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool* **where**
defined-before M L $L' \equiv$ *index* (*map lit-of* M) $L' \leq$ *index* (*map lit-of* M) L

lemma *defined-before-tl*:
assumes
L: $L' \in$ *lits-of-l* M **and**
L-*hd*: $L \neq$ *hd* (*map lit-of* M) **and**
L'-*hd*: $L' \neq$ *hd* (*map lit-of* M) **and**
def-M: *defined-before* M L L'
shows *defined-before* (*tl* M) L L'
⟨*proof*⟩

We need the following property about updates: if there is a literal L with $-L$ in the trail, and L is not watched, then it stays unwatched; i.e., while updating with *rewatch*, L does not get swapped with a watched literal L' such that $-L'$ is in the trail. This corresponds to the laziness of the data structure.

Remark that M is a trail: literals at the end were the first to be added to the trail.

primrec *watched-only-lazy-updates* :: ($'v$, $'mark$) *ann-lits* $\Rightarrow$
$'v$ *literal multiset twl-clause* $\Rightarrow$ *bool* **where**
watched-only-lazy-updates M ($TWL\text{-}Clause$ W UW) $\longleftrightarrow$
($\forall L' \in\# W. \forall L \in\# UW.$
$-L' \in$ *lits-of-l* $M \longrightarrow -L \in$ *lits-of-l* $M \longrightarrow L \notin\# W \longrightarrow$
defined-before M ($-L$) ($-L'$))

primrec *watched-wf-twl-cls-decision* **where**
watched-wf-twl-cls-decision M ($TWL\text{-}Clause$ W UW) $\longleftrightarrow$
($\forall L \in\# W. -L \in$ *lits-of-l* $M \longrightarrow$ *remove1-mset* L $W \subseteq\#$ *image-mset lit-of* (*mset* M) $\longrightarrow$
($\forall L' \in\# W. L \neq L' \longrightarrow$ *defined-before* M L' ($-L$)))

primrec *watched-wf-twl-cls-no-decision* **where**
watched-wf-twl-cls-no-decision M ($TWL\text{-}Clause$ W UW) $\longleftrightarrow$
($\forall L \in\# W. -L \in$ *lits-of-l* $M \longrightarrow \neg$*remove1-mset* L $W \subseteq\#$ *image-mset lit-of* (*mset* M) $\longrightarrow$
($\forall L' \in\# UW. L' \notin\# W \longrightarrow -L' \in$ *lits-of-l* M))

If the negation of a watched literal is included in the trail, then the negation of every unwatched literals is also included in the trail. Otherwise, the data-structure has to be updated.

fun *watched-wf-twl-cls* :: $('a, 'b)$ *ann-lits* $\Rightarrow$ *'a literal multiset twl-clause* $\Rightarrow$
bool **where**
watched-wf-twl-cls M C $\longleftrightarrow$
watched-wf-twl-cls-no-decision M C $\wedge$ *watched-wf-twl-cls-decision M C*

lemma *watched-wf-twl-cls-single-Ball*:
watched-wf-twl-cls M (TWL-Clause W UW) =
$(\forall L \in\# W. -L \in$ *lits-of-l M* $\longrightarrow$
((*remove1-mset L W* $\subseteq\#$ *image-mset lit-of (mset M)* $\longrightarrow$
$(\forall L' \in\# W. L \neq L' \longrightarrow$ *defined-before M L'* $(-L))) \wedge$
($\neg$*remove1-mset L W* $\subseteq\#$ *image-mset lit-of (mset M)* $\longrightarrow$
$(\forall L' \in\# UW. L' \notin\# W \longrightarrow -L' \in$ *lits-of-l M*))))
⟨*proof*⟩

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-twl-cls* :: $('v, 'mark)$ *ann-lits* $\Rightarrow$ *'v literal multiset twl-clause* $\Rightarrow$ *bool* **where**
wf-twl-cls M (TWL-Clause W UW) $\longleftrightarrow$
struct-wf-twl-cls (TWL-Clause W UW) $\wedge$
watched-wf-twl-cls M (TWL-Clause W UW) $\wedge$
watched-only-lazy-updates M (TWL-Clause W UW)

lemma *wf-twl-cls-annotation-independant*:
assumes *M*: *map lit-of M = map lit-of M'*
shows *wf-twl-cls M (TWL-Clause W UW)* $\longleftrightarrow$ *wf-twl-cls M' (TWL-Clause W UW)*
⟨*proof*⟩

lemma *less-eq-2-iff-eq-2-less-eq-Suc-1*: $a \leq 2 \longleftrightarrow a = 2 \vee a \leq Suc\ 0$
⟨*proof*⟩

lemma *remove-1-mset-single-add-if*:
remove1-mset K ({#L#} + *xs*) = (*if K = L then xs else* {#L#} + *remove1-mset K xs*)
⟨*proof*⟩

lemma *remove-1-mset-single-if*:
remove1-mset K {#L#} = (*if K = L then* {#} *else* {#L#})
⟨*proof*⟩

lemma *wf-twl-cls-wf-twl-cls-tl*:
fixes *C* :: *'v clause twl-clause*
assumes *wf*: *wf-twl-cls M C* **and** *n-d*: *no-dup M*
shows *wf-twl-cls (tl M) C*
⟨*proof*⟩

lemma *wf-twl-cls-append*:
assumes
n-d: *no-dup (M' @ M)* **and**
wf: *wf-twl-cls (M' @ M) C*
shows *wf-twl-cls M C*
⟨*proof*⟩

locale *well-formed-two-watched-literal-clauses-ops* =
fixes
wf-watched :: *'cls* $\Rightarrow$ *'v multiset* **and**

wf-unwatched :: *'cls* ⇒ *'v multiset*
begin

definition *wf-clause* :: *'cls* ⇒ *'v multiset* **where**
wf-clause C = *wf-watched C* + *wf-unwatched C*

fun *twl-cls-wf* :: *'cls* ⇒ *'v multiset twl-clause* **where**
twl-cls-wf C = *TWL-Clause* (*wf-watched C*) (*wf-unwatched C*)

lemma *struct-wf-twl-cls-after-switch*:
assumes
L ∈# *wf-watched C* **and**
L' ∈# *wf-unwatched C* **and**
twl-cls-wf: *struct-wf-twl-cls* (*twl-cls-wf C*)
shows
struct-wf-twl-cls
(*TWL-Clause* (*remove1-mset L* (*wf-watched C*) + {#*L'*#})
(*remove1-mset L'* (*wf-unwatched C*) + {#*L*#}))
⟨*proof*⟩
end

locale *well-formed-two-watched-literal-clauses* =
well-formed-two-watched-literal-clauses-ops wf-watched wf-unwatched
for
wf-watched :: *'cls* ⇒ *'v multiset* **and**
wf-unwatched :: *'cls* ⇒ *'v multiset* +
assumes
twl-cls-wf: *struct-wf-twl-cls* (*twl-cls-wf C*)
begin

end

experiment
begin
typedef *'v wf-twl-clause* = {*C* :: *'v multiset twl-clause*. *struct-wf-twl-cls C*}
morphisms *twl-clause-of-wf wf-of-twl-clause*
⟨*proof*⟩

setup-lifting *type-definition-wf-twl-clause*

lift-definition *wf-watched* :: *'v wf-twl-clause* ⇒ *'v multiset* **is**
watched :: *'v multiset twl-clause* ⇒ *'v multiset* ⟨*proof*⟩

lift-definition *wf-unwatched* :: *'v wf-twl-clause* ⇒ *'v multiset* **is**
unwatched :: *'v multiset twl-clause* ⇒ *'v multiset* ⟨*proof*⟩

lift-definition *wf-clause* :: *'v wf-twl-clause* ⇒ *'v multiset* **is**
clause :: *'v multiset twl-clause* ⇒ *'v multiset* ⟨*proof*⟩

lift-definition *map-wf-twl-clause* :: (*'v multiset* ⇒ *'w multiset*) ⇒ *'v wf-twl-clause* ⇒
'w multiset twl-clause **is**
map-twl-clause :: (*'v multiset* ⇒ *'w multiset*) ⇒ *'v multiset twl-clause* ⇒ *'w multiset twl-clause*
⟨*proof*⟩

lemma *wf-unwatched-wf-of-twl-clause*:
struct-wf-twl-cls C $\Longrightarrow$ *wf-unwatched* (*wf-of-twl-clause C*) = *unwatched C*
⟨*proof*⟩

lemma *wf-watched-wf-of-twl-clause*:
struct-wf-twl-cls C $\Longrightarrow$ *wf-watched* (*wf-of-twl-clause C*) = *watched C*
⟨*proof*⟩

lemma *watched-map-wf-twl-clause*:
watched (*map-wf-twl-clause f C*) = *f* (*wf-watched C*)
⟨*proof*⟩

lemma *unwatched-map-wf-twl-clause*:
unwatched (*map-wf-twl-clause f C*) = *f* (*wf-unwatched C*)
⟨*proof*⟩

lemma *wf-clause-watched-unwatched*: *wf-clause C* = *wf-watched C* + *wf-unwatched C*
⟨*proof*⟩

lemma *clause-map-wf-twl-clause-wf-clause*:
assumes $\bigwedge$*x1 x2*. *f* (*x1* + *x2*) = *f x1* + *f x2*
shows *clause* (*map-wf-twl-clause f C*) = *f* (*wf-clause C*)
⟨*proof*⟩

interpretation *well-formed-two-watched-literal-clauses-ops wf-watched wf-unwatched*
⟨*proof*⟩

interpretation *well-formed-two-watched-literal-clauses wf-watched wf-unwatched*
⟨*proof*⟩

end
end

7.3.1 Implementation for 2 Watched-Literals

theory *CDCL-Two-Watched-Literals-Implementation*
imports *CDCL-W-Abstract-State CDCL-Two-Watched-Literals*
begin

sledgehammer-params[*spy*]

The following locale axiomatizes a type introduced by the *typedef* command, by assuming that all generated theorems are true. This is necessary because one cannot create a new type in a locale.

locale *type-definition-locale* =
fixes *Abs* :: $'a \Rightarrow 'inv$ **and** *Rep* :: $'inv \Rightarrow 'a$ **and** *inv* :: $'a \Rightarrow bool$
assumes
Rep-inv: *Abs* (*Rep x*) = *x* **and**
Rep: *Rep x* $\in$ {*a. inv a*} **and**
Rep-inject: *Rep x* = *Rep y* $\longleftrightarrow$ *x* = *y* **and**
Abs-inverse: *z* $\in$ {*a. inv a*} $\Longrightarrow$ *Rep* (*Abs z*) = *z* **and**
Abs-induct: ($\bigwedge$*y. y* $\in$ {*a. inv a*} $\Longrightarrow$ *P* (*Abs y*)) $\Longrightarrow$ *P y* **and**
Rep-induct: *z* $\in$ {*a. inv a*} $\Longrightarrow$ ($\bigwedge$*z. P'* (*Rep z*)) $\Longrightarrow$ *P' z* **and**
Abs-cases: ($\bigwedge$*y. x* = *Abs y* $\Longrightarrow$ *y* $\in$ {*a. inv a*} $\Longrightarrow$ *Q*) $\Longrightarrow$ *Q* **and**
Rep-cases: *z* $\in$ {*a. inv a*} $\Longrightarrow$ ($\bigwedge$*y. z* = *Rep y* $\Longrightarrow$ *Q*) $\Longrightarrow$ *Q* **and**
Abs-inject: *z* $\in$ {*a. inv a*} $\Longrightarrow$ *z'* $\in$ {*a. inv a*} $\Longrightarrow$ *Abs z* = *Abs z'* $\longleftrightarrow$ *z* = *z'*

The difference between an implementation and the core described in the previous sections are the following:

- the candidates are cached while updating the data structure.
- instead of updating with respect to the trail only, we update with respect to the trail *and* the candidates (referred as propagate queue later).

The latter change means that we do not do the propagation as single steps where the state well-founded as described in the previous paragraph, but we do all the propagation and identify the propagation *before* the invariants hold again.

The general idea is the following:

1. Build a "propagate" queue and a conflict clause.
2. While updating the data-structure: if you find a conflicting clause, update the conflict clause. Otherwise prepend the propagated clause.
3. While updating, when looking for conflicts and propagation, work with respect to the trail of the state and the propagate queue (and not only the trail of the state).
4. As long as the propagate queue is not empty, dequeue the first element, push it on the trail (with the *conflict-driven-clause-learning*$_W$*.propagate* rule), propagate, and update the data-structure.
5. If a conflict has been found such that it is entailed by the trail only (i.e. without the propagate queue), then apply the *conflict-driven-clause-learning*$_W$*.conflict* rule.

It is important to remember that a conflicting clause with respect to the trail and the queue might not be the earliest conflicting clause, meaning that the proof of non-redundancy should not work anymore.

However, once a conflict has been found, we can stop adding literals to the queue: we just have to finish updating the data-structure (both to keep the invariant and find a potentially better conflict). A conflict is better when it involves less literals, i.e. less propagations are needed before finding the conflict.

Clauses

locale *abstract-clause-representation-ops* =
  well-formed-two-watched-literal-clauses wf-watched wf-unwatched
  for
    wf-watched :: *'cls* ⇒ *'lit multiset* **and**
    wf-unwatched :: *'cls* ⇒ *'lit multiset*
  +
  fixes
    lit-lookup :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
    lit-keys :: *'cls* ⇒ *'lit multiset* **and**

    swap-lit :: *'cls* ⇒ *'lit* ⇒ *'lit* ⇒ *'cls* **and**
    it-of-watched-ordered :: *'cls* ⇒ *'v literal* ⇒ *'lit list* **and**
    cls-of-twl-list :: *'v literal list* ⇒ *'cls*
begin

fun *map-wf-twl-clause* **where**
map-wf-twl-clause f C = TWL-Clause (f (wf-watched C)) (f (wf-unwatched C))

lemma *clause-map-wf-twl-clause-wf-clause*:
  assumes $\bigwedge$*x1 x2. f (x1 + x2) = f x1 + f x2*
  shows *clause (map-wf-twl-clause f C) = f (wf-clause C)*
  ⟨*proof*⟩

abbreviation *twl-clause* :: *'cls ⇒ 'v literal multiset twl-clause* **where**
twl-clause C ≡ map-wf-twl-clause (image-mset (λL. the (lit-lookup C L))) C

definition *clause-of-cls* :: *'cls ⇒ 'v clause* **where**
clause-of-cls C ≡ clause (twl-clause C)

lemma *wf-watched-watched-empty-iff*:
  wf-watched C = {#} ⟷ watched (twl-clause C) = {#}
  ⟨*proof*⟩

lemma *wf-watched-empty-then-wf-unwatched-empty*:
  wf-watched C = {#} ⟹ wf-unwatched C = {#}
  ⟨*proof*⟩

end

locale *abstract-clause-representation* =
  abstract-clause-representation-ops wf-watched wf-unwatched lit-lookup lit-keys swap-lit
    it-of-watched-ordered cls-of-twl-list
  for
    wf-watched :: *'cls ⇒ 'lit multiset* **and**
    wf-unwatched :: *'cls ⇒ 'lit multiset* **and**
    lit-lookup :: *'cls ⇒ 'lit ⇒ 'v literal option* **and**
    lit-keys :: *'cls ⇒ 'lit multiset* **and**

    swap-lit :: *'cls ⇒ 'lit ⇒ 'lit ⇒ 'cls* **and**
    it-of-watched-ordered :: *'cls ⇒ 'v literal ⇒ 'lit list* **and**
    cls-of-twl-list :: *'v literal list ⇒ 'cls* +
  assumes
    distinct-lit-keys[*simp*]: *distinct-mset (lit-keys C)* **and**
    valid-lit-keys: *i ∈# lit-keys C ⟷ lit-lookup C i ≠ None* **and**
    swap-lit:
      j ∈# wf-watched C ⟹ k ∈# wf-unwatched C ⟹
        twl-cls-wf (swap-lit C j k) =
          TWL-Clause
            ({#k#} + remove1-mset j (wf-watched C))
            ({#j#} + remove1-mset k (wf-unwatched C)) **and**

    it-of-watched-ordered:
      L ∈# watched (twl-clause C) ⟹
        mset (it-of-watched-ordered C L) = wf-watched C ∧
        lit-lookup C (hd (it-of-watched-ordered C L)) = Some L **and**

    twl-cls-valid: — this states that all the valid indexes are included in *C*.
      lit-keys C = wf-clause C **and**

cls-of-twl-list:
distinct D $\Longrightarrow$
clause-of-cls (*cls-of-twl-list D*) = *mset D*

begin

lemma *valid-lit-keys-SomeD*: *lit-lookup C i* = *Some e* $\Longrightarrow$ *i* $\in\#$ *lit-keys C*
⟨*proof*⟩

lemma *lit-lookup-Some-in-clause-of-cls*:
assumes *L*: *lit-lookup C i* = *Some L*
shows *L* $\in\#$ *clause-of-cls C*
⟨*proof*⟩

lemma *clause-of-cls-valid-lit-lookup*:
assumes *L*: *L* $\in\#$ *clause-of-cls C*
shows $\exists$ *i*. *lit-lookup C i* = *Some L*
⟨*proof*⟩

sublocale *abstract-with-index* **where**
get-lit = *lit-lookup* **and**
convert-to-mset = *clause-of-cls*
⟨*proof*⟩

lemma *it-of-watched-ordered-not-None*:
assumes
L: *L* $\in\#$ *watched* (*twl-clause C*) **and**
it: *it-of-watched-ordered C L* = [*j*, *k*]
shows
lit-lookup C j = *Some L* **and**
lit-lookup C k $\neq$ *None*
⟨*proof*⟩

lemma *unwatched-twl-clause-twl-cls-wff-iff*:
unwatched (*twl-clause C*) = {#} $\longleftrightarrow$ *unwatched* (*twl-cls-wf C*) = {#}
⟨*proof*⟩

lemma *distinct-plus-subset-mset-empty*:
distinct-mset (*B* + *A*) $\Longrightarrow$ *A* $\subseteq\#$ *B* $\Longrightarrow$ *A* = {#}
⟨*proof*⟩

lemma *it-of-watched-ordered-cases*:
assumes *L*: *L* $\in\#$ *watched* (*twl-clause C*)
shows
($\exists$ *j*. *it-of-watched-ordered C L* = [*j*] $\wedge$ *lit-lookup C j* = *Some L* $\wedge$
wf-unwatched C = {#} $\wedge$ *wf-watched C* = {#*j*#}) $\vee$
($\exists$ *j k*. *it-of-watched-ordered C L* = [*j*, *k*] $\wedge$ *lit-lookup C j* = *Some L* $\wedge$ *lit-lookup C k* $\neq$ *None* $\wedge$
wf-watched C = {#*j*, *k*#})
⟨*proof*⟩

end

locale *abstract-clauses-representation* =
fixes
cls-lookup :: *'clss* $\Rightarrow$ *'cls-it* $\Rightarrow$ *'cls option* **and**
cls-keys :: *'clss* $\Rightarrow$ *'cls-it multiset* **and**
clss-update :: *'clss* $\Rightarrow$ *'cls-it* $\Rightarrow$ *'cls* $\Rightarrow$ *'clss* **and**

add-cls :: *'clss* ⇒ *'cls* ⇒ *'clss* × *'cls-it*
assumes
cls-keys-distinct[*simp*]: *distinct-mset* (*cls-keys Cs*) **and**
cls-keys: $i \in\# $ *cls-keys Cs* ⟷ *cls-lookup Cs i* ≠ *None* **and**
clss-update:
cls-lookup Cs i ≠ *None* ⟹ *cls-lookup* (*clss-update Cs i C*) = (*cls-lookup Cs*) (*i* := *Some C*)
and
add-cls:
add-cls Cs C = (*Cs'*, *i*) ⟹ *cls-lookup Cs'* = (*cls-lookup Cs*) (*i* := *Some C*) **and**
add-cls-new-keys:
add-cls Cs C = (*Cs'*, *i*) ⟹ $i \notin\# $ *cls-keys Cs*
begin

lemma *add-cls-new-key*:
add-cls Cs C = (*Cs'*, *i*) ⟹ $i \in\# $ *cls-keys Cs'*
⟨*proof*⟩

abbreviation *raw-cls-of-clss* :: *'clss* ⇒ *'cls multiset* **where**
raw-cls-of-clss Cs ≡ *image-mset* (λ*L*. *the* (*cls-lookup Cs L*)) (*cls-keys Cs*)

lemma *cls-keys-clss-update*[*simp*]:
cls-lookup Cs i ≠ *None* ⟹ *cls-keys* (*clss-update Cs i E*) = *cls-keys Cs*
⟨*proof*⟩

lemma *cls-lookup-Some-in-raw-cls-of-clss*:
assumes *L*: *cls-lookup Cs i* = *Some C*
shows $C \in\# $ *raw-cls-of-clss Cs*
⟨*proof*⟩

lemma *raw-cls-of-clss-valid-cls-lookup*:
assumes *L*: $C \in\# $ *raw-cls-of-clss Cs*
shows ∃ *i*. *cls-lookup Cs i* = *Some C*
⟨*proof*⟩

sublocale *abstract-with-index2* **where**
get-lit = *cls-lookup* **and**
convert-to-mset = *raw-cls-of-clss*
⟨*proof*⟩

end

State

locale *abstract-clause-clauses-representation* =
abstract-clause-representation wf-watched wf-unwatched lit-lookup lit-keys swap-lit it-of-watched-ordered cls-of-twl-list +
abstract-clauses-representation cls-lookup cls-keys clss-update add-cls
for
wf-watched :: *'cls* ⇒ *'lit multiset* **and**
wf-unwatched :: *'cls* ⇒ *'lit multiset* **and**
lit-lookup :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
lit-keys :: *'cls* ⇒ *'lit multiset* **and**

swap-lit :: *'cls* ⇒ *'lit* ⇒ *'lit* ⇒ *'cls* **and**
it-of-watched-ordered :: *'cls* ⇒ *'v literal* ⇒ *'lit list* **and**

cls-of-twl-list :: *'v literal list* ⇒ *'cls* **and**
cls-lookup :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
cls-keys :: *'clss* ⇒ *'cls-it multiset* **and**
clss-update :: *'clss* ⇒ *'cls-it* ⇒ *'cls* ⇒ *'clss* **and**
add-cls :: *'clss* ⇒ *'cls* ⇒ *'clss* × *'cls-it*
begin

sublocale *raw-clss* **where**
get-lit = *lit-lookup* **and**
mset-cls = *clause-of-cls* **and**
get-cls = *cls-lookup* **and**
mset-clss = *raw-cls-of-clss*
⟨*proof*⟩

end

locale *abs-state*$_W$*-clss-twl-ops* =
abstract-clause-clauses-representation
wf-watched wf-unwatched
lit-lookup lit-keys swap-lit
it-of-watched-ordered cls-of-twl-list

cls-lookup cls-keys clss-update add-cls
\+
raw-cls mset-ccls
for
— Clause:
wf-watched :: *'cls* ⇒ *'lit multiset* **and**
wf-unwatched :: *'cls* ⇒ *'lit multiset* **and**
lit-lookup :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
lit-keys :: *'cls* ⇒ *'lit multiset* **and**

swap-lit :: *'cls* ⇒ *'lit* ⇒ *'lit* ⇒ *'cls* **and**
it-of-watched-ordered :: *'cls* ⇒ *'v literal* ⇒ *'lit list* **and**

— Clauses
cls-of-twl-list :: *'v literal list* ⇒ *'cls* **and**
cls-lookup :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
cls-keys :: *'clss* ⇒ *'cls-it multiset* **and**
clss-update :: *'clss* ⇒ *'cls-it* ⇒ *'cls* ⇒ *'clss* **and**
add-cls :: *'clss* ⇒ *'cls* ⇒ *'clss* × *'cls-it* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause*
begin

sublocale *abs-state*$_W$*-clss-ops* **where**
get-lit = *lit-lookup* **and**
mset-cls = *clause-of-cls* **and**
get-cls = *cls-lookup* **and**
mset-clss = *raw-cls-of-clss* **and**
mset-ccls = *mset-ccls*
⟨*proof*⟩

fun *abs-mlit* :: *'clss* ⇒ (*'v*, *'cls-it*) *ann-lit* ⇒ (*'v*, *'v clause*) *ann-lit*

where
abs-mlit Cs (*Propagated L C*) = *Propagated L* (*clause-of-cls* (*Cs* ⇓ *C*)) |
abs-mlit - (*Decided L*) = *Decided L*

lemma *lit-of-abs-mlit*[*simp*]:
lit-of (*abs-mlit Cs a*) = *lit-of a*
⟨*proof*⟩

lemma *lit-of-abs-mlit-set-lit-of-l*[*simp*]:
lit-of ‘ *abs-mlit Cs* ‘ *set M'* = *lits-of-l M'*
⟨*proof*⟩

lemma *map-abs-mlit-true-annots-true-cls*[*simp*]:
map (*abs-mlit Cs*) *M'* ⊨*as C* ⟷ *M'* ⊨*as C*
⟨*proof*⟩

end

locale *abs-state*$_W$*-twl-ops* =
abs-state$_W$*-clss-twl-ops*
— functions for clauses:
wf-watched wf-unwatched
lit-lookup lit-keys swap-lit
it-of-watched-ordered cls-of-twl-list

cls-lookup cls-keys clss-update add-cls

— functions for the conflicting clause:
mset-ccls
for
— Clause:
wf-watched :: *'cls* ⇒ *'lit multiset* **and**
wf-unwatched :: *'cls* ⇒ *'lit multiset* **and**
lit-lookup :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
lit-keys :: *'cls* ⇒ *'lit multiset* **and**

swap-lit :: *'cls* ⇒ *'lit* ⇒ *'lit* ⇒ *'cls* **and**
it-of-watched-ordered :: *'cls* ⇒ *'v literal* ⇒ *'lit list* **and**

— Clauses
cls-of-twl-list :: *'v literal list* ⇒ *'cls* **and**
cls-lookup :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
cls-keys :: *'clss* ⇒ *'cls-it multiset* **and**
clss-update :: *'clss* ⇒ *'cls-it* ⇒ *'cls* ⇒ *'clss* **and**
add-cls :: *'clss* ⇒ *'cls* ⇒ *'clss* × *'cls-it* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause* +
fixes
find-undef-in-unwatched :: *'st* ⇒ *'cls* ⇒ *'lit option* **and**
abs-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-abs-trail :: *'st* ⇒ (*'v*, *'cls-it*) *ann-lit* **and**
prop-queue :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
abs-backtrack-lvl :: *'st* ⇒ *nat* **and**

raw-conc-conflicting :: *'st* ⇒ *'ccls option* **and**

abs-learned-clss :: *'st* ⇒ *'v clauses* **and**

tl-abs-trail :: *'st* ⇒ *'st* **and**
reduce-abs-trail-to :: (*'v*, *'v clause*) *ann-lits* ⇒ *'st* ⇒ *'st* **and**

cons-prop-queue :: (*'v*, *'cls-it*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
last-prop-queue-to-trail :: *'st* ⇒ *'st* **and**
prop-queue-null :: *'st* ⇒ *bool* **and**
prop-queue-to-trail :: *'st* ⇒ *'st* **and**

add-abs-confl-to-learned-cls :: *'st* ⇒ *'st* **and**
abs-remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**

update-abs-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**

mark-conflicting :: *'cls-it* ⇒ *'st* ⇒ *'st* **and**
resolve-abs-conflicting :: *'v literal* ⇒ *'cls* ⇒ *'st* ⇒ *'st* **and**

get-undecided-lit :: *'st* ⇒ *'v literal option* **and**
get-clause-watched-by :: *'st* ⇒ *'v literal* ⇒ *'cls-it list* **and**
update-clause :: *'st* ⇒ *'cls-it* ⇒ *'cls* ⇒ *'st* **and**

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

definition *full-trail* :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **where**
full-trail S = *prop-queue S* @ *abs-trail S*

sublocale $abs\text{-}state_W\text{-}ops$ **where**
cls-lit = *lit-lookup* **and**
mset-cls = *clause-of-cls* **and**
clss-cls = *cls-lookup* **and**
mset-clss = *raw-cls-of-clss* **and**
mset-ccls = *mset-ccls* **and**

conc-trail = *full-trail* **and**
hd-raw-conc-trail = *hd-raw-abs-trail* **and**
raw-clauses = *raw-clauses* **and**
conc-backtrack-lvl = *abs-backtrack-lvl* **and**
raw-conc-conflicting = *raw-conc-conflicting* **and**
conc-learned-clss = *abs-learned-clss* **and**
cons-conc-trail = *cons-prop-queue* **and**
tl-conc-trail = λ*S*. *tl-abs-trail S* **and**
add-conc-confl-to-learned-cls = λ*S*. *add-abs-confl-to-learned-cls S* **and**
remove-cls = *abs-remove-cls* **and**
update-conc-backtrack-lvl = *update-abs-backtrack-lvl* **and**
mark-conflicting = λ*i S*. *mark-conflicting i S* **and**
reduce-conc-trail-to = λ*M S*. *reduce-abs-trail-to M* (*prop-queue-to-trail S*) **and**
resolve-conflicting = λ*L D S*. *resolve-abs-conflicting L D S* **and**
conc-init-state = *abs-init-state* **and**
restart-state = *restart-state*
⟨*proof*⟩

lemma *mmset-of-mlit-abs-mlit*[*simp*]: *mmset-of-mlit* = *abs-mlit*
⟨*proof*⟩

definition *prop-state* ::
$'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lit\ list \times ('v, 'v\ clause)\ ann\text{-}lit\ list \times 'v\ clauses \times$
$'v\ clauses \times nat \times 'v\ clause\ option$ **where**
prop-state S = (*prop-queue S*, *abs-trail S*, *conc-init-clss S*, *abs-learned-clss S*,
abs-backtrack-lvl S, *conc-conflicting S*)

lemma *prop-state-state*: *prop-state S* = $(P, M, N, U, k, C) \Longrightarrow$ *state S* = $(P\ @\ M, N, U, k, C)$
⟨*proof*⟩

end

lemma *image-mset-if-eq-index*:
$\{\#if\ x = i\ then\ P\ x\ else\ Q\ x.\ x \in\#\ M\#\} =$
$\{\#Q\ x.\ x \in\#\ removeAll\text{-}mset\ i\ M\#\} + replicate\text{-}mset\ (count\ M\ i)\ (P\ i)$ (**is** *?M M* = -)
⟨*proof*⟩

locale *abs-state*$_W$*-twl* =
abs-state$_W$*-twl-ops*
— functions for clauses:
wf-watched wf-unwatched
lit-lookup lit-keys swap-lit
it-of-watched-ordered cls-of-twl-list

cls-lookup cls-keys clss-update add-cls

— functions for the conflicting clause:
mset-ccls

find-undef-in-unwatched

abs-trail hd-raw-abs-trail prop-queue raw-clauses abs-backtrack-lvl raw-conc-conflicting

abs-learned-clss

tl-abs-trail reduce-abs-trail-to

cons-prop-queue last-prop-queue-to-trail prop-queue-null prop-queue-to-trail

add-abs-confl-to-learned-cls abs-remove-cls

update-abs-backtrack-lvl mark-conflicting resolve-abs-conflicting

get-undecided-lit get-clause-watched-by update-clause

abs-init-state restart-state

for
— Clause:
wf-watched :: $'cls \Rightarrow 'lit\ multiset$ **and**
wf-unwatched :: $'cls \Rightarrow 'lit\ multiset$ **and**
lit-lookup :: $'cls \Rightarrow 'lit \Rightarrow 'v\ literal\ option$ **and**
lit-keys :: $'cls \Rightarrow 'lit\ multiset$ **and**

$swap\text{-}lit :: 'cls \Rightarrow 'lit \Rightarrow 'lit \Rightarrow 'cls$ **and**
$it\text{-}of\text{-}watched\text{-}ordered :: 'cls \Rightarrow 'v\ literal \Rightarrow 'lit\ list$ **and**

— Clauses
$cls\text{-}of\text{-}twl\text{-}list :: 'v\ literal\ list \Rightarrow 'cls$ **and**
$cls\text{-}lookup :: 'clss \Rightarrow 'cls\text{-}it \Rightarrow 'cls\ option$ **and**
$cls\text{-}keys :: 'clss \Rightarrow 'cls\text{-}it\ multiset$ **and**
$clss\text{-}update :: 'clss \Rightarrow 'cls\text{-}it \Rightarrow 'cls \Rightarrow 'clss$ **and**
$add\text{-}cls :: 'clss \Rightarrow 'cls \Rightarrow 'clss \times 'cls\text{-}it$ **and**

— Conflicting clause:
$mset\text{-}ccls :: 'ccls \Rightarrow 'v\ clause$ **and**

$find\text{-}undef\text{-}in\text{-}unwatched :: 'st \Rightarrow 'cls \Rightarrow 'lit\ option$ **and**

$abs\text{-}trail :: 'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lits$ **and**
$hd\text{-}raw\text{-}abs\text{-}trail :: 'st \Rightarrow ('v, 'cls\text{-}it)\ ann\text{-}lit$ **and**
$prop\text{-}queue :: 'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lits$ **and**
$raw\text{-}clauses :: 'st \Rightarrow 'clss$ **and**
$abs\text{-}backtrack\text{-}lvl :: 'st \Rightarrow nat$ **and**
$raw\text{-}conc\text{-}conflicting :: 'st \Rightarrow 'ccls\ option$ **and**

$abs\text{-}learned\text{-}clss :: 'st \Rightarrow 'v\ clauses$ **and**

$tl\text{-}abs\text{-}trail :: 'st \Rightarrow 'st$ **and**
$reduce\text{-}abs\text{-}trail\text{-}to :: ('v, 'v\ clause)\ ann\text{-}lits \Rightarrow 'st \Rightarrow 'st$ **and**

$cons\text{-}prop\text{-}queue :: ('v, 'cls\text{-}it)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$last\text{-}prop\text{-}queue\text{-}to\text{-}trail :: 'st \Rightarrow 'st$ **and**
$prop\text{-}queue\text{-}null :: 'st \Rightarrow bool$ **and**
$prop\text{-}queue\text{-}to\text{-}trail :: 'st \Rightarrow 'st$ **and**

$add\text{-}abs\text{-}confl\text{-}to\text{-}learned\text{-}cls :: 'st \Rightarrow 'st$ **and**
$abs\text{-}remove\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**

$update\text{-}abs\text{-}backtrack\text{-}lvl :: nat \Rightarrow 'st \Rightarrow 'st$ **and**

$mark\text{-}conflicting :: 'cls\text{-}it \Rightarrow 'st \Rightarrow 'st$ **and**
$resolve\text{-}abs\text{-}conflicting :: 'v\ literal \Rightarrow 'cls \Rightarrow 'st \Rightarrow 'st$ **and**

$get\text{-}undecided\text{-}lit :: 'st \Rightarrow 'v\ literal\ option$ **and**
$get\text{-}clause\text{-}watched\text{-}by :: 'st \Rightarrow 'v\ literal \Rightarrow 'cls\text{-}it\ list$ **and**
$update\text{-}clause :: 'st \Rightarrow 'cls\text{-}it \Rightarrow 'cls \Rightarrow 'st$ **and**

$abs\text{-}init\text{-}state :: 'clss \Rightarrow 'st$ **and**
$restart\text{-}state :: 'st \Rightarrow 'st$ +

assumes
$prop\text{-}state\text{-}cons\text{-}prop\text{-}queue$:
$\bigwedge T'.\ undefined\text{-}lit\ (full\text{-}trail\ T)\ (lit\text{-}of\ L) \Longrightarrow$
$prop\text{-}state\ T = (P,\ T') \Longrightarrow valid\text{-}annotation\ T\ L \Longrightarrow$
$prop\text{-}state\ (cons\text{-}prop\text{-}queue\ L\ T) = (abs\text{-}mlit\ (raw\text{-}clauses\ T)\ L\ \#\ P,\ \ T')$ **and**

$last\text{-}prop\text{-}queue\text{-}to\text{-}trail\text{-}prop\text{-}state$:
$\bigwedge T'.\ prop\text{-}queue\ T \neq [] \Longrightarrow$
$prop\text{-}state\ T = (P,\ M,\ T') \Longrightarrow$

prop-state (*last-prop-queue-to-trail* T) =
(*but-last* P, *last* P # M, T') **and**
prop-queue-to-trail-prop-state:
$\bigwedge T'$. *prop-state* T = (P, M, T') $\Longrightarrow$
prop-state (*prop-queue-to-trail* T) = ([], P @ M, T') **and**
raw-conc-conflicting-prop-queue-to-trail[*simp*]:
raw-conc-conflicting (*prop-queue-to-trail* *st*) = *raw-conc-conflicting* *st* **and**
raw-clauses-prop-queue-to-trail[*simp*]:
raw-clauses (*prop-queue-to-trail* *st*) = *raw-clauses* *st* **and**

hd-raw-abs-trail:
full-trail *st* $\neq$ [] $\Longrightarrow$
mmset-of-mlit (*raw-clauses* *st*) (*hd-raw-abs-trail* *st*) = *hd* (*full-trail* *st*) **and**

tl-abs-trail-prop-state:
$\bigwedge S'$. *prop-state* *st* = (P, M, S') $\Longrightarrow$
prop-state (*tl-abs-trail* *st*) = (*tl* P, *if* P = [] *then* *tl* M *else* M, S') **and**

abs-remove-cls:
$\bigwedge S'$. *prop-state* *st* = (P, M, N, U, S') $\Longrightarrow$
prop-state (*abs-remove-cls* C' *st*) =
(P, M, *removeAll-mset* (*clause-of-cls* C') N, *removeAll-mset* (*clause-of-cls* C') U, S') **and**

add-abs-confl-to-learned-cls:
no-dup (*full-trail* *st*) $\Longrightarrow$ *prop-state* *st* = (P, M, N, U, k, *Some* F) $\Longrightarrow$
prop-state (*add-abs-confl-to-learned-cls* *st*) =
(P, M, N, {#F#} + U, k, *None*) **and**

update-abs-backtrack-lvl:
$\bigwedge S'$. *prop-state* *st* = (P, M, N, U, k, S') $\Longrightarrow$
prop-state (*update-abs-backtrack-lvl* k' *st*) = (P, M, N, U, k', S') **and**

mark-conflicting-prop-state:
prop-state *st* = (P, M, N, U, k, *None*) $\Longrightarrow$ E $\in\Downarrow$ *raw-clauses* *st* $\Longrightarrow$
prop-state (*mark-conflicting* E *st*) =
(P, M, N, U, k, *Some* (*clause-of-cls* (*raw-clauses* *st* $\Downarrow$ E)))
and

resolve-abs-conflicting:
prop-state *st* = (P, M, N, U, k, *Some* F) $\Longrightarrow$ $-L'$ $\in$# F $\Longrightarrow$ L' $\in$# *clause-of-cls* D $\Longrightarrow$
prop-state (*resolve-abs-conflicting* L' D *st*) =
(P, M, N, U, k, *Some* (*cdcl*$_W$*-mset.resolve-cls* L' F (*clause-of-cls* D))) **and**

prop-state-abs-init-state:
prop-state (*abs-init-state* *Ns*) = ([], [], *clauses-of-clss* *Ns*, {#}, *0*, *None*) **and**

— Properties about restarting *restart-state*:
prop-queue-restart-state[*simp*]: *prop-queue* (*restart-state* S) = [] **and**
abs-trail-restart-state[*simp*]: *abs-trail* (*restart-state* S) = [] **and**
conc-init-clss-restart-state[*simp*]: *conc-init-clss* (*restart-state* S) = *conc-init-clss* S **and**
abs-learned-clss-restart-state[*intro*]:
abs-learned-clss (*restart-state* S) $\subseteq$# *abs-learned-clss* S **and**
abs-backtrack-lvl-restart-state[*simp*]: *abs-backtrack-lvl* (*restart-state* S) = *0* **and**
conc-conflicting-restart-state[*simp*]: *conc-conflicting* (*restart-state* S) = *None* **and**

— Properties about *reduce-abs-trail-to*:

reduce-abs-trail-to:
$\bigwedge S'.$ *abs-trail st* $=$ *M2* @ *M1* $\Longrightarrow$ *prop-state st* $= ([], M, S') \Longrightarrow$
prop-state (*reduce-abs-trail-to M1 st*) $= ([], M1, S')$ **and**

learned-clauses:
abs-learned-clss S $\subseteq\#$ *conc-clauses S* **and**

get-undecided-lit-Some:
get-undecided-lit T $=$ *Some* $L' \Longrightarrow$ *undefined-lit* (*abs-trail T*) $L' \wedge$
atm-of $L' \in$ *atms-of-mm* (*conc-clauses T*) **and**
get-undecided-lit-None:
get-undecided-lit T $=$ *None* $\longleftrightarrow$
($\forall L'.$ *atm-of* $L' \in$ *atms-of-mm* (*conc-clauses T*) $\longrightarrow \neg$*undefined-lit* (*abs-trail T*) L') **and**
get-clause-watched-by:
$i \in$ *set* (*get-clause-watched-by T K*) $\longleftrightarrow$ ($K \in\#$ *watched* (*twl-clause* (*raw-clauses T* $\Downarrow$ *i*)) $\wedge$
$i \in\Downarrow$ *raw-clauses S*) **and**
get-clause-watched-by-distinct:
distinct (*get-clause-watched-by T K*) **and**

update-clause:
$i \in\Downarrow$ *raw-clauses S* $\Longrightarrow$
raw-clauses (*update-clause S i E'*) $=$ *clss-update* (*raw-clauses S*) *i E'* **and**
update-clause-state:
$i \in\Downarrow$ *raw-clauses S* $\Longrightarrow$ *prop-state S* $= (P, M, N, U, k, C) \Longrightarrow$
prop-state (*update-clause S i E'*) $= (P, M,$ *conc-init-clss S*, *abs-learned-clss S*, $k, C)$ **and**

find-undef-in-unwatched-Some:
find-undef-in-unwatched S E' $=$ *Some j* $\Longrightarrow j \in\downarrow E' \wedge$ *undefined-lit* (*full-trail S*) $(E'\downarrow j) \wedge$
$(E'\downarrow j) \in\#$ *unwatched* (*twl-clause E'*) **and**
find-undef-in-unwatched-None:
find-undef-in-unwatched S E' $=$ *None* $\longleftrightarrow$
$(\forall j.\ j \in\downarrow E' \longrightarrow (E'\downarrow j) \in\#$ *unwatched* (*twl-clause E'*) $\longrightarrow$
$\neg$*undefined-lit* (*full-trail S*) $(E'\downarrow j)$) **and**

prop-queue-null[*iff*]:
prop-queue-null S $\longleftrightarrow$ *List.null* (*prop-queue S*)
begin

lemma
prop-queue-prop-queue-to-trail[*simp*]:
prop-queue (*prop-queue-to-trail S*) $= []$ **and**
abs-trail-prop-queue-to-trail[*simp*]:
abs-trail (*prop-queue-to-trail S*) $=$ *prop-queue S* @ *abs-trail S* **and**
full-trail-prop-queue-to-trail[*simp*]:
full-trail (*prop-queue-to-trail S*) $=$ *prop-queue S* @ *abs-trail S* **and**
conc-init-clss-prop-queue-to-trail[*simp*]:
conc-init-clss (*prop-queue-to-trail S*) $=$ *conc-init-clss S* **and**
abs-learned-clss-prop-queue-to-trail[*simp*]:
abs-learned-clss (*prop-queue-to-trail S*) $=$ *abs-learned-clss S* **and**
abs-backtrack-lvl-prop-queue-to-trail[*simp*]:
abs-backtrack-lvl (*prop-queue-to-trail S*) $=$ *abs-backtrack-lvl S* **and**
conc-conflicting-prop-queue-to-trail[*simp*]:
conc-conflicting (*prop-queue-to-trail S*) $=$ *conc-conflicting S*
$\langle$*proof*$\rangle$

lemma

shows
abs-trail-tl-abs-trail[*simp*]:
prop-queue (*tl-abs-trail* S) = *tl* (*prop-queue* S) **and**
full-trail-tl-abs-trail[*simp*]:
full-trail (*tl-abs-trail* S) = *tl* (*full-trail* S) **and**
conc-init-clss-tl-abs-trail[*simp*]:
conc-init-clss (*tl-abs-trail* S) = *conc-init-clss* S **and**
abs-learned-clss-tl-abs-trail[*simp*]:
abs-learned-clss (*tl-abs-trail* S) = *abs-learned-clss* S **and**
abs-backtrack-lvl-tl-abs-trail[*simp*]:
abs-backtrack-lvl (*tl-abs-trail* S) = *abs-backtrack-lvl* S **and**
conc-conflicting-tl-abs-trail[*simp*]:
conc-conflicting (*tl-abs-trail* S) = *conc-conflicting* S
⟨*proof*⟩

lemma
assumes *raw-conc-conflicting* S = *Some* F **and** *no-dup* (*full-trail* S)
shows
prop-queue-add-abs-confl-to-learned-cls[*simp*]:
prop-queue (*add-abs-confl-to-learned-cls* S) = *prop-queue* S **and**
abs-trail-add-abs-confl-to-learned-cls[*simp*]:
abs-trail (*add-abs-confl-to-learned-cls* S) = *abs-trail* S **and**
full-trail-add-abs-confl-to-learned-cls[*simp*]:
full-trail (*add-abs-confl-to-learned-cls* S) = *full-trail* S **and**
conc-init-clss-add-abs-confl-to-learned-cls[*simp*]:
conc-init-clss (*add-abs-confl-to-learned-cls* S) = *conc-init-clss* S **and**
abs-learned-clss-add-abs-confl-to-learned-cls[*simp*]:
abs-learned-clss (*add-abs-confl-to-learned-cls* S) = {#*mset-ccls* F#} + *abs-learned-clss* S **and**
abs-backtrack-lvl-add-abs-confl-to-learned-cls[*simp*]:
abs-backtrack-lvl (*add-abs-confl-to-learned-cls* S) = *abs-backtrack-lvl* S **and**
conc-conflicting-add-abs-confl-to-learned-cls[*simp*]:
conc-conflicting (*add-abs-confl-to-learned-cls* S) = *None*
⟨*proof*⟩

lemma *state-cons-prop-queue*:
assumes
undef: *undefined-lit* (*full-trail st*) (*lit-of* L) **and**
st: *state st* = (M, S') **and**
valid-annotation st L
shows *state* (*cons-prop-queue* L *st*) = (*mmset-of-mlit* (*raw-clauses st*) L # M, S')
⟨*proof*⟩

lemma *cons-conc-trail*:
assumes *state st* = (M, S')
shows *state* (*tl-abs-trail st*) = (*tl* M, S')
⟨*proof*⟩

lemma *remove-cls*:
assumes *state st* = (M, N, U, S')
shows *state* (*abs-remove-cls* C *st*) =
(M, *removeAll-mset* (*clause-of-cls* C) N, *removeAll-mset* (*clause-of-cls* C) U, S')
⟨*proof*⟩

lemma *add-conc-confl-to-learned-cls*:
assumes *no-dup* (*full-trail st*) **and**
state st = (M, N, U, k, *Some* F)

shows *state* (*add-abs-confl-to-learned-cls st*) = (M, N, {#F#} + U, k, *None*)
⟨*proof*⟩

lemma *mark-conflicting*:
assumes
state st = (M, N, U, k, *None*) **and**
E ∈⇓ *raw-clauses st*
shows *state* (*mark-conflicting E st*) =
(M, N, U, k, *Some* (*clause-of-cls* (*raw-clauses st* ⇓ E)))
⟨*proof*⟩

lemma *abs-init-state*:
state (*abs-init-state Ns*) = ([], *clauses-of-clss Ns*, {#}, *0*, *None*)
⟨*proof*⟩

lemma *reduce-conc-trail-to*:
assumes
full-trail st = *M2* @ *M1* **and**
state st = (M, S')
shows *state* (*reduce-abs-trail-to M1* (*prop-queue-to-trail st*)) = (*M1*, S')
⟨*proof*⟩

lemma *resolve-conflicting*:
assumes
state st = (M, N, U, k, *Some F*) **and**
− L' ∈# F **and**
L' ∈# *clause-of-cls D*
shows *state* (*resolve-abs-conflicting* L' *D st*) =
(M, N, U, k, *Some* (*remove1-mset* (− L') F #∪ *remove1-mset* L' (*clause-of-cls D*)))
⟨*proof*⟩

sublocale *abs-state*$_W$ **where**
cls-lit = *lit-lookup* **and**
mset-cls = *clause-of-cls* **and**
clss-cls = *cls-lookup* **and**
mset-clss = *raw-cls-of-clss* **and**
mset-ccls = *mset-ccls* **and**

conc-trail = *full-trail* **and**
hd-raw-conc-trail = *hd-raw-abs-trail* **and**
raw-clauses = *raw-clauses* **and**
conc-backtrack-lvl = *abs-backtrack-lvl* **and**
raw-conc-conflicting = *raw-conc-conflicting* **and**
conc-learned-clss = *abs-learned-clss* **and**
cons-conc-trail = *cons-prop-queue* **and**
tl-conc-trail = λS. *tl-abs-trail S* **and**
add-conc-confl-to-learned-cls = λS. *add-abs-confl-to-learned-cls S* **and**
remove-cls = *abs-remove-cls* **and**
update-conc-backtrack-lvl = *update-abs-backtrack-lvl* **and**
mark-conflicting = λi S. *mark-conflicting i S* **and**
reduce-conc-trail-to = λM S. *reduce-abs-trail-to M* (*prop-queue-to-trail S*) **and**
resolve-conflicting = λL D S. *resolve-abs-conflicting L D S* **and**
conc-init-state = *abs-init-state* **and**
restart-state = *restart-state*
⟨*proof*⟩

lemma *image-mset-mset-remove1*: $a \in\# B \Longrightarrow$
$\{\#f\ x.\ x \in\#\ \mathit{remove1\text{-}mset}\ a\ B\#\} = \mathit{remove1\text{-}mset}\ (f\ a)\ \{\#f\ x.\ x \in\# B\#\}$
⟨*proof*⟩

lemma *distinct-disinst-mset-incl-iff-set-incl*:
$\mathit{distinct}\ A \Longrightarrow \mathit{distinct}\ B \Longrightarrow \mathit{mset}\ A \subseteq\#\ \mathit{mset}\ B \longleftrightarrow \mathit{set}\ A \subseteq \mathit{set}\ B$
⟨*proof*⟩

lemma *conc-clauses-update-clause*:
assumes
i: $i \in\Downarrow \mathit{raw\text{-}clauses}\ S$
shows
$\mathit{conc\text{-}clauses}\ (\mathit{update\text{-}clause}\ S\ i\ E) =$
$\mathit{remove1\text{-}mset}\ (\mathit{clause\text{-}of\text{-}cls}\ (\mathit{raw\text{-}clauses}\ S \Downarrow i))\ (\mathit{conc\text{-}clauses}\ S) + \{\#\mathit{clause\text{-}of\text{-}cls}\ E\#\}$
(**is** *?abs* = *?r*)
⟨*proof*⟩

definition *wf-prop-queue* :: $'st \Rightarrow \mathit{bool}$ **where**
$\mathit{wf\text{-}prop\text{-}queue}\ S \longleftrightarrow (\forall M \in \mathit{set}\ (\mathit{prop\text{-}queue}\ S).\ \mathit{is\text{-}proped}\ M)$

function *all-annotation-valid* **where**
$\mathit{all\text{-}annotation\text{-}valid}\ S \longleftrightarrow$
$(\mathit{if}\ \mathit{full\text{-}trail}\ S = []$
$\mathit{then}\ \mathit{True}$
$\mathit{else}\ \mathit{valid\text{-}annotation}\ S\ (\mathit{hd\text{-}raw\text{-}abs\text{-}trail}\ S) \wedge \mathit{all\text{-}annotation\text{-}valid}\ (\mathit{tl\text{-}abs\text{-}trail}\ S))$
⟨*proof*⟩
termination
⟨*proof*⟩

declare *all-annotation-valid.simps*[*simp del*]

lemma *all-annotation-valid-simps*[*simp*]:
shows
$\mathit{full\text{-}trail}\ S = [] \Longrightarrow \mathit{all\text{-}annotation\text{-}valid}\ S$ **and**
$\mathit{full\text{-}trail}\ S \neq [] \Longrightarrow \mathit{all\text{-}annotation\text{-}valid}\ S = (\mathit{valid\text{-}annotation}\ S\ (\mathit{hd\text{-}raw\text{-}abs\text{-}trail}\ S)$
$\wedge\ \mathit{all\text{-}annotation\text{-}valid}\ (\mathit{tl\text{-}abs\text{-}trail}\ S))$
⟨*proof*⟩

definition *wf-twl-state* :: $'st \Rightarrow \mathit{bool}$ **where**
$\mathit{wf\text{-}twl\text{-}state}\ S \longleftrightarrow$
$(\mathit{full\text{-}trail}\ S \neq [] \longrightarrow \mathit{all\text{-}annotation\text{-}valid}\ S) \wedge$
$\mathit{cdcl}_W\mathit{\text{-}mset.cdcl}_W\mathit{\text{-}all\text{-}struct\text{-}inv}\ (\mathit{state}\ S) \wedge$
$\mathit{wf\text{-}prop\text{-}queue}\ S$

end

The new Calculus

fun *reduce-trail-to-lvl* :: $\mathit{nat} \Rightarrow ('a,\ 'b)\ \mathit{ann\text{-}lit}\ \mathit{list} \Rightarrow ('a,\ 'b)\ \mathit{ann\text{-}lit}\ \mathit{list}$ **where**
$\mathit{reduce\text{-}trail\text{-}to\text{-}lvl}\ \text{-}\ [] = []\ |$
$\mathit{reduce\text{-}trail\text{-}to\text{-}lvl}\ \mathit{target\text{-}lvl}\ (\mathit{Decided}\ L\ \#\ M) =$
$(\mathit{if}\ \mathit{count\text{-}decided}\ M = \mathit{target\text{-}lvl}\ \mathit{then}\ M$
$\mathit{else}\ \mathit{reduce\text{-}trail\text{-}to\text{-}lvl}\ \mathit{target\text{-}lvl}\ M)\ |$
$\mathit{reduce\text{-}trail\text{-}to\text{-}lvl}\ \mathit{lvl}\ (\mathit{Propagated}\ L\ C\ \#\ M) = \mathit{reduce\text{-}trail\text{-}to\text{-}lvl}\ \mathit{lvl}\ M$

fun *reduce-trail-to-lvl-no-count* :: $\mathit{nat} \Rightarrow \mathit{nat} \Rightarrow ('a,\ 'b)\ \mathit{ann\text{-}lit}\ \mathit{list} \Rightarrow ('a,\ 'b)\ \mathit{ann\text{-}lit}\ \mathit{list}$ **where**

reduce-trail-to-lvl-no-count - - [] = [] |
reduce-trail-to-lvl-no-count target-lvl current-lvl (*Decided L* # *M*) =
  (*if Suc target-lvl* = *current-lvl then M*
  else reduce-trail-to-lvl-no-count target-lvl (*current-lvl* − *1*) *M*) |
  — *Suc lvl* is the level we are seeking plus one, *Suc current-lvl* is the current level.
reduce-trail-to-lvl-no-count lvl current-lvl (*Propagated L C* # *M*) =
  reduce-trail-to-lvl-no-count lvl current-lvl M

lemma *reduce-trail-to-lvl-reduce-trail-to-lvl-no-count*:
  reduce-trail-to-lvl i M = *reduce-trail-to-lvl-no-count i* (*count-decided M*) *M*
  ⟨*proof*⟩

lemma *reduce-trail-to-lvl-lvl-eq*[*simp*]:
  reduce-trail-to-lvl (*count-decided M* − *1*) *M* = *tl* (*dropWhile* (λL. ¬*is-decided L*) *M*)
  ⟨*proof*⟩

lemma *reduce-trail-to-lvl-lvl-ge*:
  i ≥ *count-decided M* ⟹ *reduce-trail-to-lvl i M* = []
  ⟨*proof*⟩

lemma *reduce-trail-to-lvl-lvl-ge-lvl*:
  reduce-trail-to-lvl i M = [] ⟹ *i* ≥ *count-decided M* ∨ *i* = *0*
  ⟨*proof*⟩

lemma *reduce-trail-to-lvl-decomp-lvl*:
  assumes *i* < *count-decided M*
  shows (∃ *M′ L*. *M* = *M′* @ *Decided L* # *reduce-trail-to-lvl i M*) ∧
    count-decided (*reduce-trail-to-lvl i M*) = *i*
  ⟨*proof*⟩

lemma *reduce-trail-to-lvl-skip-not-marked-at-beginning*:
  assumes ∀ *m* ∈ *set M′*. ¬*is-decided m*
  shows *reduce-trail-to-lvl i* (*M′* @ *Decided L* # *M″*) = *reduce-trail-to-lvl i* (*Decided L* # *M″*)
  ⟨*proof*⟩

lemma *count-decided-tl-dropWhile-not-decided*:
  count-decided (*tl* (*dropWhile* (λL. ¬ *is-decided L*) *M*)) = *count-decided M* − *1*
  ⟨*proof*⟩

locale *abs-conflict-driven-clause-learning*$_W$*-clss* =
  abs-state$_W$*-twl*
    — functions for clauses:
    wf-watched wf-unwatched
    lit-lookup lit-keys swap-lit
    it-of-watched-ordered cls-of-twl-list

    cls-lookup cls-keys clss-update add-cls

    — functions for the conflicting clause:
    mset-ccls

    find-undef-in-unwatched

    abs-trail hd-raw-abs-trail prop-queue raw-clauses abs-backtrack-lvl raw-conc-conflicting *abs-learned-clss*

tl-abs-trail reduce-abs-trail-to

cons-prop-queue last-prop-queue-to-trail prop-queue-null prop-queue-to-trail

add-abs-confl-to-learned-cls abs-remove-cls

update-abs-backtrack-lvl mark-conflicting resolve-abs-conflicting

get-undecided-lit get-clause-watched-by update-clause

abs-init-state restart-state +
type-definition-locale
abs-state-of rough-state-of wf-twl-state
for
— Clause:
wf-watched :: *'cls* ⇒ *'lit multiset* **and**
wf-unwatched :: *'cls* ⇒ *'lit multiset* **and**
lit-lookup :: *'cls* ⇒ *'lit* ⇒ *'v literal option* **and**
lit-keys :: *'cls* ⇒ *'lit multiset* **and**

swap-lit :: *'cls* ⇒ *'lit* ⇒ *'lit* ⇒ *'cls* **and**
it-of-watched-ordered :: *'cls* ⇒ *'v literal* ⇒ *'lit list* **and**

— Clauses
cls-of-twl-list :: *'v literal list* ⇒ *'cls* **and**
cls-lookup :: *'clss* ⇒ *'cls-it* ⇒ *'cls option* **and**
cls-keys :: *'clss* ⇒ *'cls-it multiset* **and**
clss-update :: *'clss* ⇒ *'cls-it* ⇒ *'cls* ⇒ *'clss* **and**
add-cls :: *'clss* ⇒ *'cls* ⇒ *'clss* × *'cls-it* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause* **and**

find-undef-in-unwatched :: *'st* ⇒ *'cls* ⇒ *'lit option* **and**

abs-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-abs-trail :: *'st* ⇒ (*'v*, *'cls-it*) *ann-lit* **and**
prop-queue :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
abs-backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conc-conflicting :: *'st* ⇒ *'ccls option* **and**

abs-learned-clss :: *'st* ⇒ *'v clauses* **and**

tl-abs-trail :: *'st* ⇒ *'st* **and**
reduce-abs-trail-to :: (*'v*, *'v clause*) *ann-lits* ⇒ *'st* ⇒ *'st* **and**

cons-prop-queue :: (*'v*, *'cls-it*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
last-prop-queue-to-trail :: *'st* ⇒ *'st* **and**
prop-queue-null :: *'st* ⇒ *bool* **and**
prop-queue-to-trail :: *'st* ⇒ *'st* **and**

add-abs-confl-to-learned-cls :: *'st* ⇒ *'st* **and**
abs-remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**

update-abs-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**

mark-conflicting :: *'cls-it* ⇒ *'st* ⇒ *'st* **and**
resolve-abs-conflicting :: *'v literal* ⇒ *'cls* ⇒ *'st* ⇒ *'st* **and**

get-undecided-lit :: *'st* ⇒ *'v literal option* **and**
get-clause-watched-by :: *'st* ⇒ *'v literal* ⇒ *'cls-it list* **and**
update-clause :: *'st* ⇒ *'cls-it* ⇒ *'cls* ⇒ *'st* **and**

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

abs-state-of :: *'st* ⇒ *'inv* **and**
rough-state-of :: *'inv* ⇒ *'st*
begin

sublocale *abs-conflict-driven-clause-learning*$_W$ **where**
get-lit = *lit-lookup* **and**
mset-cls = *clause-of-cls* **and**
get-cls = *cls-lookup* **and**
mset-clss = *raw-cls-of-clss* **and**
mset-ccls = *mset-ccls* **and**

conc-trail = *full-trail* **and**
hd-raw-conc-trail = *hd-raw-abs-trail* **and**
raw-clauses = *raw-clauses* **and**
conc-backtrack-lvl = *abs-backtrack-lvl* **and**
raw-conc-conflicting = *raw-conc-conflicting* **and**
conc-learned-clss = *abs-learned-clss* **and**
cons-conc-trail = *cons-prop-queue* **and**
tl-conc-trail = λ*S*. *tl-abs-trail S* **and**
add-conc-confl-to-learned-cls = λ*S*. *add-abs-confl-to-learned-cls S* **and**
remove-cls = *abs-remove-cls* **and**
update-conc-backtrack-lvl = *update-abs-backtrack-lvl* **and**
mark-conflicting = λ*i S*. *mark-conflicting i S* **and**
reduce-conc-trail-to = λ*M S*. *reduce-abs-trail-to M* (*prop-queue-to-trail S*) **and**
resolve-conflicting = λ*L D S*. *resolve-abs-conflicting L D S* **and**
conc-init-state = *abs-init-state* **and**
restart-state = *restart-state*
⟨*proof*⟩

lemma *XXX*: *type-definition rough-state-of abs-state-of* {*S*. *wf-twl-state S*}
⟨*proof*⟩

definition *wf-resolve* :: *'inv* ⇒ *'inv* ⇒ *bool* **where**
wf-resolve S T ≡ *resolve-abs* (*rough-state-of S*) (*rough-state-of T*)

abbreviation *mark-conflicting-and-flush* **where**
mark-conflicting-and-flush i S ≡ *mark-conflicting i* (*prop-queue-to-trail S*)

fun *is-of-maximum-level* :: *'v clause* ⇒ (*'v*, *'b*) *ann-lit list* ⇒ *bool* **where**
is-of-maximum-level C [] ⟷ *True* |
is-of-maximum-level C (*Decided L'* # *M*) ⟷ −*L'* ∉# *C* |
is-of-maximum-level C (*Propagated L'* - # *M*) ⟷ −*L'* ∉# *C* ∧ *is-of-maximum-level C M*

lemma *is-of-maximum-level-decomposition*:
  assumes *is-of-maximum-level C M*
  shows
    $\exists$ M' L' M''. (($M = M'$ @ *Decided* L' # M'' $\wedge$ $-L' \notin\#$ C) $\vee$ ($M = M' \wedge M'' = []$)) $\wedge$
    ($\forall$ $m \in$ *set* M'. $\neg$*is-decided* m) $\wedge$
    uminus ‘ *set-mset* C $\cap$ *lits-of-l* $M' = \{\}$
  ⟨*proof*⟩

lemma *true-annots-CNot-uminus-incl-iff*:
  M $\models$*as CNot* C $\longleftrightarrow$ *uminus* ‘ *set-mset* C $\subseteq$ *lits-of-l* M
  ⟨*proof*⟩

lemma *get-maximum-level-skip-Decide-first*:
  assumes *atm-of* $L \notin$ *atms-of* D **and** *atms-of* $D \subseteq$ *atm-of* ‘ *lits-of-l* M
  shows *get-maximum-level* (*Decided* L # M) D = *get-maximum-level* M D
  ⟨*proof*⟩

The following lemma gives the relation between *is-of-maximum-level* and the inequality on the level. The clause C is expected to be instantiated by a clause like *remove1-mset* L (*mset-ccls* E), where E is the conflicting clause.

lemma
  fixes M :: ($'v$, $'b$) *ann-lits* **and** L :: $'v$ *literal* **and** D :: $'b$
  defines LM[*simp*]: $LM \equiv$ *Propagated* L D # M
  assumes
    n-d: *no-dup* LM **and**
    max: *is-of-maximum-level* C M **and**
    M-C: LM $\models$*as CNot* C **and**
    L-C: $-L \notin\#$ C
  shows
    get-maximum-level (*Propagated* L D # M) C < *count-decided* (*Propagated* L D # M) $\vee$ $C = \{\#\}$
⟨*proof*⟩

definition *wf-state* :: $'st \Rightarrow 'inv$ **where**
wf-state S = *abs-state-of* (*if* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state* S) *then* S *else* S)

lemma [*code abstype*]:
  wf-state (*rough-state-of* S) = S
  ⟨*proof*⟩

fun *backtrack-implementation* **where**
backtrack-implementation S =
  reduce-abs-trail-to (*reduce-trail-to-lvl* (*abs-backtrack-lvl* S) (*full-trail* S))

function (*domintros*) *skip-or-resolve* **where**
skip-or-resolve S =
  (*if full-trail* S = [] *then* S
  else
    case hd-raw-abs-trail S *of*
      Decided $L \Rightarrow S$
    | *Propagated* L C $\Rightarrow$
      if $-L \in\#$ *mset-ccls* (*the* (*raw-conc-conflicting* S))
      then
        if is-of-maximum-level (*mset-ccls* (*the* (*raw-conc-conflicting* S))) (*tl* (*full-trail* S))
        then S
        else skip-or-resolve (*tl-abs-trail* (*resolve-abs-conflicting* L (*raw-clauses* S $\Downarrow$ C) S))

else skip-or-resolve (*tl-abs-trail S*))
⟨*proof*⟩

lemma
assumes
$cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*) **and**
raw-conc-conflicting $S \neq None$
shows *skip-or-resolve-dom S*
⟨*proof*⟩

When we update a clause with respect to the literal L, there are several cases:

1. the only literal is L: this is a conflict.
2. if the other watched literal is true, there is noting to do.
3. if it is false, then we have found a conflict (since every unwatched literal has to be false).
4. otherwise, we have to check if we can find a literal to swap or propagate the variable.

fun *update-watched-clause* :: $'st \Rightarrow 'v$ *literal* $\Rightarrow$ $'cls\text{-}it \Rightarrow 'st$ **where**
update-watched-clause S L i =
(*case it-of-watched-ordered* (*raw-clauses S* ⇓ *i*) *L of*
[-] ⇒ *mark-conflicting i S*
| [*j*, *k*] ⇒
if ((*raw-clauses S* ⇓ *i*) ↓ *k*) ∈ *lits-of-l* (*abs-trail S*)
then S
else if −((*raw-clauses S* ⇓ *i*) ↓ *k*) ∈ *lits-of-l* (*abs-trail S*)
then mark-conflicting i S
else
(*case find-undef-in-unwatched S* (*raw-clauses S* ⇓ *i*) *of*
None ⇒ *cons-prop-queue* (*Propagated L i*) *S*
| *Some* - ⇒ *update-clause S i* (*swap-lit* (*raw-clauses S* ⇓ *i*) *j k*))
)

lemma
fixes *i* :: *'cls-it* **and** *S* :: *'st* **and** *L* :: *'v literal*
defines *S'*: *S'* ≡ *update-watched-clause S L i*
assumes
$cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*) **and**
L: *L* ∈# *watched* (*twl-clause* (*raw-clauses S* ⇓ *i*)) **and**
confl: *raw-conc-conflicting S* = *None* **and**
i: *i* ∈⇓ *raw-clauses S* **and**
L-trail: − *L* ∈ *lits-of-l* (*full-trail S*)
shows *propagate-abs S S'* ∨ *conflict-abs S S'*
⟨*proof*⟩

Possible optimisation: *Option.is-none* (*raw-conc-conflicting S'*) is the same as checking whether conflict has been marked by *update-watched-clause*.

fun *update-watched-clauses* :: *'st* ⇒ *'v literal* ⇒ *'cls-it list* ⇒ *'st* **where**
update-watched-clauses S L (*i* # *Cs*) =
(*let S'* = *update-watched-clause S L i in*
if Option.is-none (*raw-conc-conflicting S'*)
then update-watched-clauses S' L Cs
else S') |

update-watched-clauses S L [] = *S*

definition *propagate-and-conflict-one-lit* **where**
propagate-and-conflict-one-lit S L =
  update-watched-clauses S L (*get-clause-watched-by S L*)

lemma *raw-conc-conflicting-mark-conflicting*:
  assumes *i* ∈⇓ *raw-clauses S* **and** *raw-conc-conflicting S* = *None*
  shows *raw-conc-conflicting* (*mark-conflicting i S*) ≠ *None*
  ⟨*proof*⟩

lemma
  assumes *Option.is-none* (*raw-conc-conflicting S*) **and** −*L* ∈ *lits-of-l* (*full-trail S*)
  shows
    state S = *state* (*propagate-and-conflict-one-lit S L*) ∨
    conflict-abs S (*propagate-and-conflict-one-lit S L*)
  ⟨*proof*⟩

function (*domintros*) *propagate-and-conflict* **where**
propagate-and-conflict S =
  (*if prop-queue-null S*
  then S
  else
    let S′ = *prop-queue-to-trail S in*
    propagate-and-conflict (*propagate-and-conflict-one-lit S′* (*lit-of* (*hd-raw-abs-trail S′*))))
⟨*proof*⟩

end

end
theory *CDCL-Two-Watched-Literals-Implementation-RBT*
imports *Main RBT-More CDCL-Abstract-Clause-Representation CDCL-W-Level*
  CDCL-Two-Watched-Literals CDCL-Two-Watched-Literals-Implementation
begin

interpretation *raw-clss* **where**
  get-lit = *RBT.lookup* **and**
  mset-cls = λ*C*. *mset* (*map snd* (*RBT.entries C*)) **and**
  get-cls = *RBT.lookup* **and**
  mset-clss = λ*C*. *mset* (*map snd* (*RBT.entries C*))
  ⟨*proof*⟩

definition *get-unwatched* :: (*nat*, ′*b*) *RBT.rbt* ⇒ ′*b multiset* **where**
get-unwatched C = *mset* (*map snd* (*filter* (λ*L*. *fst L* ≥ *2*) (*RBT.entries C*)))

definition *get-watched* :: (*nat*, ′*b*) *RBT.rbt* ⇒ ′*b multiset* **where**
get-watched C =
  (*let append-if-not-None* =
    (λ*i*. *case RBT.lookup C i of None* ⇒ *op* + {#} | *Some a* ⇒ *op* + {#*a*#}) *in*
    append-if-not-None 0 (*append-if-not-None 1* {#}))

lemma *ge-Suc-Suc-0-iff*: *a* ≥ *Suc* (*Suc 0*) ⟷ *a* ≠ *0* ∧ *a* ≠ *Suc 0*
  ⟨*proof*⟩

lemma *less-2-iff*: *n* < *2* ⟷ *n* = *0* ∨ *n* = *Suc 0*

⟨*proof*⟩

lemma *count-RBT-entries*:
count (*mset* (*RBT.entries C*)) (*a*, *b*) = (*if RBT.lookup C a* = *Some b then 1 else 0*)
⟨*proof*⟩

lemma *filter-RBT-entries-le-2*:
{# *x* ∈# *mset* (*RBT.entries C*). *fst x* < (*2*::*nat*)#} =
(*if RBT.lookup C 0* ≠ *None then* {#(*0*, *the* (*RBT.lookup C 0*))#} *else* {#}) +
(*if RBT.lookup C 1* ≠ *None then* {#(*1*, *the* (*RBT.lookup C 1*))#} *else* {#})
⟨*proof*⟩

Gere is another definition of *get-watched*, analog to *get-unwatched*:

lemma *get-watched-map-le-2*:
get-watched C = *mset* (*map snd* (*filter* (λ*L*. *fst L* < *2*) (*RBT.entries C*)))
⟨*proof*⟩

definition *get-watched-list* :: (*nat*, ′*b*) *RBT.rbt* ⇒ ′*b list* **where**
get-watched-list C =
(*let append-if-not-None* =
(λ*i*. *case RBT.lookup C i of None* ⇒ *id* | *Some a* ⇒ *op* # *a*) *in*
append-if-not-None 0 (*append-if-not-None 1* []))

definition *clause-of-RBT* :: (*nat*, ′*a*) *RBT.rbt* ⇒ ′*a multiset twl-clause* **where**
clause-of-RBT C = *TWL-Clause* (*get-watched C*) (*get-unwatched C*)

typedef ′*v wf-clause-RBT* =
{*C* :: (*nat*, ′*v*) *RBT.rbt*. *struct-wf-twl-cls* (*clause-of-RBT C*)}
morphisms *conc-RBT-cls abs-RBT-cls*
⟨*proof*⟩

fun *RBT-clause* :: (*nat*, ′*a*) *RBT.rbt* ⇒ ′*a multiset* **where**
RBT-clause C = *get-watched C* + *get-unwatched C*

setup-lifting *type-definition-wf-clause-RBT*
lift-definition *wf-watched* :: ′*v wf-clause-RBT* ⇒ ′*v multiset* **is** *get-watched* ⟨*proof*⟩
lift-definition *wf-watched-list* :: ′*v wf-clause-RBT* ⇒ ′*v list* **is** *get-watched-list* ⟨*proof*⟩
lift-definition *wf-unwatched* :: ′*v wf-clause-RBT* ⇒ ′*v multiset* **is** *get-unwatched* ⟨*proof*⟩
lift-definition *lit-lookup* :: ′*v wf-clause-RBT* ⇒ *nat* ⇀ ′*v* **is** *RBT.lookup* ⟨*proof*⟩
lift-definition *lit-keys* :: ′*v wf-clause-RBT* ⇒ *nat multiset* **is** λ*C*. *mset* (*RBT.keys C*) ⟨*proof*⟩
lift-definition *wf-RBT-clause* :: ′*v wf-clause-RBT* ⇒ ′*v multiset* **is** *RBT-clause* ⟨*proof*⟩

The following function is a bit more general than needed: we only call it when i and j are well-formed indexes.

fun *swap-lit-safe* :: (′*a*::*linorder*, ′*b*) *RBT.rbt* ⇒ ′*a* ⇒ ′*a* ⇒ (′*a*, ′*b*) *RBT.rbt* **where**
swap-lit-safe C i j =
(*case* (*RBT.lookup C i*, *RBT.lookup C j*) *of*
(*Some i*′, *Some j*′) ⇒ *RBT.insert j i*′ (*RBT.insert i j*′ *C*)
| - ⇒ *C*)

interpretation *well-formed-two-watched-literal-clauses-ops* **where**
wf-watched = *wf-watched* **and**

wf-unwatched = *wf-unwatched*
⟨*proof*⟩

interpretation *raw-RBT-clause*: *well-formed-two-watched-literal-clauses-ops* **where**
wf-watched = *get-watched* **and**
wf-unwatched = *get-unwatched*
⟨*proof*⟩

interpretation *well-formed-two-watched-literal-clauses* **where**
wf-watched = *wf-watched* **and**
wf-unwatched = *wf-unwatched*
⟨*proof*⟩

lemma *mset-entries-map-snd-insert*:
$P\ (j,\ j') \Longrightarrow P\ (j,\ i') \Longrightarrow RBT.lookup\ C\ j = Some\ j' \Longrightarrow$
$mset\ (map\ snd\ [L{\leftarrow}RBT.entries\ (RBT.insert\ j\ i'\ C)\ .\ P\ L]) =$
$\{\#i'\#\} + remove1\text{-}mset\ j'\ (mset\ (map\ snd\ [L{\leftarrow}RBT.entries\ C\ .\ P\ L]))$
⟨*proof*⟩

lemma *clause-of-RBT-swap-lit-safe-commute-index*:
$clause\text{-}of\text{-}RBT\ (swap\text{-}lit\text{-}safe\ C\ i\ j) = clause\text{-}of\text{-}RBT\ (swap\text{-}lit\text{-}safe\ C\ j\ i)$
⟨*proof*⟩

lemma *image-mset-snd-remove1-mset-entries*:
$RBT.lookup\ C\ j = Some\ j' \Longrightarrow P\ (j,\ j') \Longrightarrow$
$image\text{-}mset\ snd$
$(remove1\text{-}mset\ (j,\ j')$
$\{\#\ x \in\#\ mset\ (RBT.entries\ C).\ P\ x\#\}) =$
$remove1\text{-}mset\ j'$
$(image\text{-}mset\ snd$
$\{\#\ x \in\#\ mset\ (RBT.entries\ C).\ P\ x\#\})$
⟨*proof*⟩

lemma *clause-of-RBT-swap-lit-safe*:
assumes $i \le j$ **and** $struct\text{-}wf\text{-}twl\text{-}cls\ (TWL\text{-}Clause\ (get\text{-}watched\ C)\ (get\text{-}unwatched\ C))$
shows $clause\text{-}of\text{-}RBT\ (swap\text{-}lit\text{-}safe\ C\ i\ j) =$
$(case\ (RBT.lookup\ C\ i,\ RBT.lookup\ C\ j)\ of$
$(Some\ i',\ Some\ j') \Rightarrow$
$if\ i < 2 \wedge j < 2$
$then\ clause\text{-}of\text{-}RBT\ C$
$else\ if\ i < 2 \wedge j \ge 2$
$then\ TWL\text{-}Clause\ (\{\#j'\#\} + remove1\text{-}mset\ i'\ (get\text{-}watched\ C))$
$(\{\#i'\#\} + remove1\text{-}mset\ j'\ (get\text{-}unwatched\ C))$
$else\ clause\text{-}of\text{-}RBT\ C$
$|\ \text{-} \Rightarrow clause\text{-}of\text{-}RBT\ C)$
⟨*proof*⟩

lift-definition *swap-lit* :: $'v\ wf\text{-}clause\text{-}RBT \Rightarrow nat \Rightarrow nat \Rightarrow\ 'v\ wf\text{-}clause\text{-}RBT$ **is**
swap-lit-safe
⟨*proof*⟩

fun *it-of-watched-ordered* :: $'a\ wf\text{-}clause\text{-}RBT \Rightarrow\ 'a \Rightarrow\ 'a\ list$ **where**
$it\text{-}of\text{-}watched\text{-}ordered\ C\ L =$
$(case\ wf\text{-}watched\text{-}list\ C\ of$
$[\text{-}] \Rightarrow [L]$
$|\ [L1,\ L2] \Rightarrow if\ L1 = L\ then\ [L1,\ L2]\ else\ [L2,\ L1])$

fun *list-to-RBT* :: *'a list* ⇒ *nat* ⇒ (*nat*, *'a*) *RBT.rbt* **where**
list-to-RBT [] - = *RBT.empty* |
list-to-RBT (*L* # *C*) *n* = *RBT.insert n L* (*list-to-RBT C* (*Suc n*))

The following functions works only if there are no duplicate in *C*. Otherwise, the result is not specified.

fun *cls-of-twl-list* :: *'a list* ⇒ *'a wf-clause-RBT* **where**
cls-of-twl-list C = *abs-RBT-cls* (*list-to-RBT C 0*)

lemma *RBT-lookup-list-to-RBT*:
RBT.lookup (*list-to-RBT C i*) *j* = (*if j* ≥ *i* ∧ *j* < *i* + *length C then Some* (*C* ! (*j* − *i*)) *else None*)
⟨*proof*⟩

lemma *mset-RBT-entries-list-to-RBT*:
mset (*RBT.entries* (*list-to-RBT C i*)) = *mset* (*zip* [*i*..< *i*+*length C*] *C*)
⟨*proof*⟩

lemma *mset-zip-image-mset*:
mset (*zip xs ys*) = {# (*xs*!*i*, *ys*!*i*). *i* ∈# *mset* [*0*..< *min* (*length xs*) (*length ys*)] #}
⟨*proof*⟩

lemma *mset-set-eq*:
finite A ⟹ *finite B* ⟹ *mset-set A* = *mset-set B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *filter-image-mset*:
{# *L* ∈# {#*P x*. *x* ∈# *M*#}. *Q L*#} = {#*P x*| *x* ∈# *M*. *Q* (*P x*)#}
⟨*proof*⟩

lemma *image-mset-mset-mset-map*:
image-mset f (*mset l*) = *mset* (*map f l*)
⟨*proof*⟩

lemma *image-mset-nth-upt*:
image-mset (*op* ! *C*) (*mset-set* {*0*..<*length C*}) = *mset C*
⟨*proof*⟩

lemma *image-mset-snd-mset-RBT-entries*:
image-mset snd (*mset* (*RBT.entries* (*list-to-RBT C 0*))) = *mset C*
⟨*proof*⟩

lemma *wf-RBT-clause-cls-of-twl-list*:
assumes *dist-C*: *distinct C*
shows *wf-RBT-clause* (*cls-of-twl-list C*) = *mset C*
⟨*proof*⟩

lemma *mset-RBT-entries*:
mset (*map snd* (*RBT.entries C*)) = *get-watched C* + *get-unwatched C*
⟨*proof*⟩

fun *twl-clause-of-rbt* **where**
twl-clause-of-rbt C =
(*let append-if-not-None* =
(λ*i*. *case RBT.lookup C i of None* ⇒ *id* | *Some a* ⇒ *Cons a*) *in*
TWL-Clause (*get-watched C*)

$(get\text{-}unwatched\ C))$

lemma
assumes $i \in set\ (RBT.keys\ C)$ **and** $j \in set\ (RBT.keys\ C)$
shows $RBT.lookup\ (swap\text{-}lit\text{-}safe\ C\ j\ i) =$
$RBT.lookup\ C(j \mapsto the\ (RBT.lookup\ C\ i),\ i \mapsto the\ (RBT.lookup\ C\ j))$
$\langle proof \rangle$

end