Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

June 21, 2016

Contents

Chapter 1

More Standard Theorems

This chapter contains additional lemmas built on top of HOL.

end

theory *Multiset-More*
imports ~~/*src/HOL/Library/Multiset-Order*
begin

1.1 More about Multisets

Isabelle's theory of finite multisets is not as developed as other areas, such as lists and sets. The present theory introduces some missing concepts and lemmas. Some of it is expected to move to Isabelle's library.

1.1.1 Basic Setup

declare
diff-single-trivial [*simp*]
in-image-mset [*iff*]
image-mset.compositionality [*simp*]

mset-leD[*dest*, *intro?*]

Multiset.in-multiset-in-set[*simp*]

lemma *image-mset-cong2*[*cong*]:
$(\bigwedge x.\ x \in\# M \Longrightarrow f\ x = g\ x) \Longrightarrow M = N \Longrightarrow$ *image-mset f M* = *image-mset g N*
⟨*proof*⟩

lemma *subset-msetE* [*elim*!]:
[|*A* ⊂# *B*; [|*A* ⊆# *B*; ~ (*B*⊆#*A*)|] ==> *R*|] ==> *R*
⟨*proof*⟩

1.1.2 Lemmas about intersections

lemma *mset-inter-single*:

$x \in\# \Sigma \Longrightarrow \Sigma \#\cap \{\#x\#\} = \{\#x\#\}$
$x \notin\# \Sigma \Longrightarrow \Sigma \#\cap \{\#x\#\} = \{\#\}$
⟨*proof*⟩

1.1.3 Lemmas about size

This sections adds various lemmas about size. Most lemmas have a finite set equivalent.

lemma *size-mset-SucE*: $size\ A = Suc\ n \Longrightarrow (\bigwedge a\ B.\ A = \{\#a\#\} + B \Longrightarrow size\ B = n \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *size-Suc-Diff1*:
$x \in\# \Sigma \Longrightarrow Suc\ (size\ (\Sigma - \{\#x\#\})) = size\ \Sigma$
⟨*proof*⟩

lemma *size-Diff-singleton*: $x \in\# \Sigma \Longrightarrow size\ (\Sigma - \{\#x\#\}) = size\ \Sigma - 1$
⟨*proof*⟩

lemma *size-Diff-singleton-if*: $size\ (A - \{\#x\#\}) = (if\ x \in\# A\ then\ size\ A - 1\ else\ size\ A)$
⟨*proof*⟩

lemma *size-Un-Int*:
$size\ A + size\ B = size\ (A \#\cup B) + size\ (A \#\cap B)$
⟨*proof*⟩

lemma *size-Un-disjoint*:
assumes $A \#\cap B = \{\#\}$
shows $size\ (A \#\cup B) = size\ A + size\ B$
⟨*proof*⟩

lemma *size-Diff-subset-Int*:
shows $size\ (\Sigma - \Sigma') = size\ \Sigma - size\ (\Sigma \#\cap \Sigma')$
⟨*proof*⟩

lemma *diff-size-le-size-Diff*: $size\ (\Sigma::\ -\ multiset) - size\ \Sigma' \leq size\ (\Sigma - \Sigma')$
⟨*proof*⟩

lemma *size-Diff1-less*: $x \in\# \Sigma \Longrightarrow size\ (\Sigma - \{\#x\#\}) < size\ \Sigma$
⟨*proof*⟩

lemma *size-Diff2-less*: $x \in\# \Sigma \Longrightarrow y \in\# \Sigma \Longrightarrow size\ (\Sigma - \{\#x\#\} - \{\#y\#\}) < size\ \Sigma$
⟨*proof*⟩

lemma *size-Diff1-le*: $size\ (\Sigma - \{\#x\#\}) \leq size\ \Sigma$
⟨*proof*⟩

lemma *size-psubset*: $(\Sigma::\ -\ multiset) \leq\# \Sigma' \Longrightarrow size\ \Sigma < size\ \Sigma' \Longrightarrow \Sigma <\# \Sigma'$
⟨*proof*⟩

1.1.4 Multiset Extension of Multiset Ordering

The *op* $\#\subset\#\#$ and *op* $\#\subseteq\#\#$ operators are introduced as the multiset extension of the multiset orderings of *op* $\#\subset\#$ and *op* $\#\subseteq\#$.

definition *less-mset-mset* :: $('a :: order)\ multiset\ multiset \Rightarrow 'a\ multiset\ multiset \Rightarrow bool$
(**infix** $\#<\#\#\ 50$)

where
$M' \#<\#\# M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' \#<\# x\}$

definition *le-mset-mset* :: $('a :: order)\ multiset\ multiset \Rightarrow 'a\ multiset\ multiset \Rightarrow bool$
(**infix** $\#<=\#\#\ 50$)
where
$M' \#<=\#\# M \longleftrightarrow M' \#<\#\# M \vee M' = M$

notation *less-mset-mset* (**infix** $\#\subset\#\#\ 50$)
notation *le-mset-mset* (**infix** $\#\subseteq\#\#\ 50$)

lemmas $less\text{-}mset\text{-}mset_{DM} = order.mult_{DM}[OF\ order\text{-}multiset, folded\ less\text{-}mset\text{-}mset\text{-}def]$
lemmas $less\text{-}mset\text{-}mset_{HO} = order.mult_{HO}[OF\ order\text{-}multiset, folded\ less\text{-}mset\text{-}mset\text{-}def]$

interpretation *multiset-multiset-order*: *order*
le-mset-mset :: $('a :: linorder)\ multiset\ multiset \Rightarrow ('a :: linorder)\ multiset\ multiset \Rightarrow bool$
less-mset-mset :: $('a :: linorder)\ multiset\ multiset \Rightarrow ('a::linorder)\ multiset\ multiset \Rightarrow bool$
⟨*proof*⟩

interpretation *multiset-multiset-linorder*: *linorder*
le-mset-mset :: $('a :: linorder)\ multiset\ multiset \Rightarrow ('a :: linorder)\ multiset\ multiset \Rightarrow bool$
less-mset-mset :: $('a :: linorder)\ multiset\ multiset \Rightarrow ('a::linorder)\ multiset\ multiset \Rightarrow bool$
⟨*proof*⟩

lemma *wf-less-mset-mset*: $wf\ \{(\Sigma :: ('a :: wellorder)\ multiset\ multiset,\ T).\ \Sigma \#\subset\#\#\ T\}$
⟨*proof*⟩

interpretation *multiset-multiset-wellorder*: *wellorder*
le-mset-mset :: $('a::wellorder)\ multiset\ multiset \Rightarrow ('a::wellorder)\ multiset\ multiset \Rightarrow bool$
less-mset-mset :: $('a::wellorder)\ multiset\ multiset \Rightarrow ('a::wellorder)\ multiset\ multiset \Rightarrow bool$
⟨*proof*⟩

lemma *union-less-mset-mset-mono2*: $B\ \#\subset\#\#\ D \Longrightarrow C + B\ \#\subset\#\#\ C + (D::'a::order\ multiset\ multiset)$
⟨*proof*⟩

lemma *union-less-mset-mset-diff-plus*:
$U \leq\# \Sigma \Longrightarrow T\ \#\subset\#\#\ U \Longrightarrow \Sigma - U + T\ \#\subset\#\#\ \Sigma$
⟨*proof*⟩

lemma *ex-gt-imp-less-mset-mset*:
$(\exists\, y :: 'a :: linorder\ multiset \in\#\ T.\ (\forall\, x.\ x \in\#\ \Sigma \longrightarrow x\ \#\subset\#\ y)) \Longrightarrow \Sigma\ \#\subset\#\#\ T$
⟨*proof*⟩

1.1.5 Remove

lemma *set-mset-minus-replicate-mset*[*simp*]:
$n \geq count\ A\ a \Longrightarrow set\text{-}mset\ (A - replicate\text{-}mset\ n\ a) = set\text{-}mset\ A - \{a\}$
$n < count\ A\ a \Longrightarrow set\text{-}mset\ (A - replicate\text{-}mset\ n\ a) = set\text{-}mset\ A$
⟨*proof*⟩

abbreviation *removeAll-mset* :: $'a \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **where**
$removeAll\text{-}mset\ C\ M \equiv M - replicate\text{-}mset\ (count\ M\ C)\ C$

lemma *mset-removeAll*[*simp*, *code*]:
$removeAll\text{-}mset\ C\ (mset\ L) = mset\ (removeAll\ C\ L)$

⟨*proof*⟩

lemma *removeAll-mset-filter-mset*:
removeAll-mset C M = *filter-mset* (*op* ≠ *C*) *M*
⟨*proof*⟩

abbreviation *remove1-mset* :: *'a* ⇒ *'a multiset* ⇒ *'a multiset* **where**
remove1-mset C M ≡ *M* − {#*C*#}

lemma *remove1-mset-remove1*[*code*]:
remove1-mset C (*mset L*) = *mset* (*remove1 C L*)
⟨*proof*⟩

lemma *in-remove1-mset-neq*:
assumes *ab*: *a* ≠ *b*
shows *a* ∈# *remove1-mset b C* ⟷ *a* ∈# *C*
⟨*proof*⟩

lemma *size-mset-removeAll-mset-le-iff*:
size (*removeAll-mset x M*) < *size M* ⟷ *x* ∈# *M*
⟨*proof*⟩

lemma *size-mset-remove1-mset-le-iff*:
size (*remove1-mset x M*) < *size M* ⟷ *x* ∈# *M*
⟨*proof*⟩

lemma *set-mset-remove1-mset*[*simp*]:
set-mset (*remove1-mset L* (*mset W*)) = *set* (*remove1 L W*)
⟨*proof*⟩

1.1.6 Replicate

lemma *replicate-mset-plus*: *replicate-mset* (*a* + *b*) *C* = *replicate-mset a C* + *replicate-mset b C*
⟨*proof*⟩

lemma *mset-replicate-replicate-mset*:
mset (*replicate n L*) = *replicate-mset n L*
⟨*proof*⟩

lemma *set-mset-single-iff-replicate-mset*:
set-mset U = {*a*} ⟷ (∃ *n*>*0*. *U* = *replicate-mset n a*) (**is** *?S* ⟷ *?R*)
⟨*proof*⟩

1.1.7 Multiset and set conversion

lemma *count-mset-set-if*:
count (*mset-set A*) *a* = (*if a* ∈ *A* ∧ *finite A then 1 else 0*)
⟨*proof*⟩

lemma *mset-set-set-mset-empty-mempty*[*iff*]:
mset-set (*set-mset D*) = {#} ⟷ *D* = {#}
⟨*proof*⟩

lemma *size-mset-set-card*:
finite S ⟹ *size* (*mset-set S*) = *card S*
⟨*proof*⟩

lemma *count-mset-set-le-one*: *count* (*mset-set A*) *x* ≤ *1*
⟨*proof*⟩

lemma *mset-set-subseteq-mset-set*[*iff*]:
assumes *finite A finite B*
shows *mset-set A* ⊆# *mset-set B* ⟷ *A* ⊆ *B*
⟨*proof*⟩

lemma *mset-set-set-mset-subseteq*[*simp*]: *mset-set* (*set-mset A*) ⊆# *A*
⟨*proof*⟩

lemma *mset-sorted-list-of-set*[*simp*]:
mset (*sorted-list-of-set A*) = *mset-set A*
⟨*proof*⟩

lemma *mset-take-subseteq*: *mset* (*take n xs*) ⊆# *mset xs*
⟨*proof*⟩

1.1.8 Removing duplicates

definition *remdups-mset* :: *'v multiset* ⇒ *'v multiset* **where**
remdups-mset S = *mset-set* (*set-mset S*)

lemma *remdups-mset-in*[*iff*]: *a* ∈# *remdups-mset A* ⟷ *a* ∈# *A*
⟨*proof*⟩

lemma *count-remdups-mset-eq-1*: *a* ∈# *remdups-mset A* ⟷ *count* (*remdups-mset A*) *a* = *1*
⟨*proof*⟩

lemma *remdups-mset-empty*[*simp*]:
remdups-mset {#} = {#}
⟨*proof*⟩

lemma *remdups-mset-singleton*[*simp*]:
remdups-mset {#*a*#} = {#*a*#}
⟨*proof*⟩

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

lemma *remdups-mset-eq-empty*[*iff*]:
remdups-mset D = {#} ⟷ *D* = {#}
⟨*proof*⟩

lemma *remdups-mset-singleton-sum*[*simp*]:
remdups-mset ({#*a*#} + *A*) = (*if a* ∈# *A then remdups-mset A else* {#*a*#} + *remdups-mset A*)
remdups-mset (*A*+{#*a*#}) = (*if a* ∈# *A then remdups-mset A else* {#*a*#} + *remdups-mset A*)
⟨*proof*⟩

lemma *mset-remdups-remdups-mset*[*simp*]:
mset (*remdups D*) = *remdups-mset* (*mset D*)
⟨*proof*⟩

definition *distinct-mset* :: *'a multiset* ⇒ *bool* **where**
distinct-mset S ⟷ (∀ *a*. *a* ∈# *S* ⟶ *count S a* = *1*)

lemma *distinct-mset-empty*[*simp*]: *distinct-mset* {#}
⟨*proof*⟩

lemma *distinct-mset-singleton*[*simp*]: *distinct-mset* {#*a*#}
⟨*proof*⟩

definition *distinct-mset-set* :: *'a multiset set* ⇒ *bool* **where**
distinct-mset-set Σ ⟷ (∀ *S* ∈Σ. *distinct-mset S*)

lemma *distinct-mset-set-empty*[*simp*]:
distinct-mset-set {}
⟨*proof*⟩

lemma *distinct-mset-set-singleton*[*iff*]:
distinct-mset-set {*A*} ⟷ *distinct-mset A*
⟨*proof*⟩

lemma *distinct-mset-set-insert*[*iff*]:
distinct-mset-set (*insert S* Σ) ⟷ (*distinct-mset S* ∧ *distinct-mset-set* Σ)
⟨*proof*⟩

lemma *distinct-mset-set-union*[*iff*]:
distinct-mset-set (Σ ∪ Σ′) ⟷ (*distinct-mset-set* Σ ∧ *distinct-mset-set* Σ′)
⟨*proof*⟩

lemma *distinct-mset-union*:
assumes *dist*: *distinct-mset* (*A* + *B*)
shows *distinct-mset A*
⟨*proof*⟩

lemma *distinct-mset-minus*[*simp*]:
distinct-mset A ⟹ *distinct-mset* (*A* − *B*)
⟨*proof*⟩

lemma *in-distinct-mset-set-distinct-mset*:
a ∈ Σ ⟹ *distinct-mset-set* Σ ⟹ *distinct-mset a*
⟨*proof*⟩

lemma *distinct-mset-remdups-mset*[*simp*]: *distinct-mset* (*remdups-mset S*)
⟨*proof*⟩

lemma *distinct-mset-distinct*[*simp*]:
distinct-mset (*mset x*) = *distinct x*
⟨*proof*⟩

lemma *distinct-mset-mset-set*:
distinct-mset (*mset-set A*)
⟨*proof*⟩

lemma *distinct-mset-rempdups-union-mset*:
assumes *distinct-mset A* **and** *distinct-mset B*
shows *A* #∪ *B* = *remdups-mset* (*A* + *B*)
⟨*proof*⟩

lemma *distinct-mset-set-distinct*:

distinct-mset-set (*mset* ‘ *set Cs*) ⟷ (∀ *c*∈ *set Cs*. *distinct c*)
⟨*proof*⟩

lemma *distinct-mset-add-single*:
distinct-mset ({#*a*#} + *L*) ⟷ *distinct-mset L* ∧ *a* ∉# *L*
⟨*proof*⟩

lemma *distinct-mset-single-add*:
distinct-mset (*L* + {#*a*#}) ⟷ *distinct-mset L* ∧ *a* ∉# *L*
⟨*proof*⟩

lemma *distinct-mset-size-eq-card*:
distinct-mset C ⟹ *size C* = *card* (*set-mset C*)
⟨*proof*⟩

Another characterisation of *distinct-mset*

lemma *distinct-mset-count-less-1*:
distinct-mset S ⟷ (∀ *a*. *count S a* ≤ *1*)
⟨*proof*⟩

lemma *distinct-mset-add*:
distinct-mset (*L* + *L*′) ⟷ *distinct-mset L* ∧ *distinct-mset L*′ ∧ *L* #∩ *L*′ = {#} (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

lemma *distinct-mset-set-mset-ident*[*simp*]: *distinct-mset M* ⟹ *mset-set* (*set-mset M*) = *M*
⟨*proof*⟩

lemma *distinct-finite-set-mset-subseteq-iff*[*iff*]:
assumes *dist*: *distinct-mset M* **and** *fin*: *finite N*
shows *set-mset M* ⊆ *N* ⟷ *M* ⊆# *mset-set N*
⟨*proof*⟩

lemma *distinct-mem-diff-mset*:
assumes *dist*: *distinct-mset M* **and** *mem*: *x* ∈ *set-mset* (*M* − *N*)
shows *x* ∉ *set-mset N*
⟨*proof*⟩

lemma *distinct-set-mset-eq*:
assumes
dist-m: *distinct-mset M* **and**
dist-n: *distinct-mset N* **and**
set-eq: *set-mset M* = *set-mset N*
shows *M* = *N*
⟨*proof*⟩

lemma *distinct-mset-union-mset*:
assumes
distinct-mset D **and**
distinct-mset C
shows *distinct-mset* (*D* #∪ *C*)
⟨*proof*⟩

lemma *distinct-mset-inter-mset*:
assumes
distinct-mset D **and**
distinct-mset C

shows *distinct-mset* $(D \#\cap C)$
⟨*proof*⟩

lemma *distinct-mset-remove1-All*:
distinct-mset $C \Longrightarrow$ *remove1-mset* $L\ C =$ *removeAll-mset* $L\ C$
⟨*proof*⟩

lemma *distinct-mset-size-2*: *distinct-mset* $\{\#a, b\#\} \longleftrightarrow a \neq b$
⟨*proof*⟩

1.1.9 Filter

lemma *mset-filter-compl*: *mset* (*filter* $p\ xs$) + *mset* (*filter* $(Not \circ p)\ xs$) = *mset* xs
⟨*proof*⟩

lemma *image-mset-subseteq-mono*: $A \subseteq\# B \Longrightarrow$ *image-mset* $f\ A \subseteq\#$ *image-mset* $f\ B$
⟨*proof*⟩

lemma *image-filter-ne-mset*[*simp*]:
image-mset $f\ \{\#x \in\# M.\ f\ x \neq y\#\} =$ *removeAll-mset* y (*image-mset* $f\ M$)
⟨*proof*⟩

lemma *comprehension-mset-False*[*simp*]:
$\{\#\ L \in\#\ A.\ False\#\} = \{\#\}$
⟨*proof*⟩

Near duplicate of *filter-eq-replicate-mset*: $\{\#\ y \in\#\ ?D.\ y = ?x\#\}$ = *replicate-mset* (*count* *?D* *?x*) *?x*.

lemma *filter-mset-eq*:
filter-mset $(op = L)\ A =$ *replicate-mset* (*count* $A\ L$) L
⟨*proof*⟩

lemma *filter-mset-union-mset*:
filter-mset $P\ (A \#\cup B) =$ *filter-mset* $P\ A \#\cup$ *filter-mset* $P\ B$
⟨*proof*⟩

lemma *filter-mset-mset-set*:
finite $A \Longrightarrow$ *filter-mset* P (*mset-set* A) = *mset-set* $\{a \in A.\ P\ a\}$
⟨*proof*⟩

See *filter-cong* for the set version. Mark as [*fundef-cong*] too?

lemma *filter-mset-cong*:
assumes [*simp*]: $M = M'$ **and** [*simp*]: $\bigwedge a.\ a \in\# M \Longrightarrow P\ a = Q\ a$
shows *filter-mset* $P\ M =$ *filter-mset* $Q\ M$
⟨*proof*⟩

1.1.10 Sums

lemma *msetsum-distrib*[*simp*]:
fixes $C\ D :: 'a \Rightarrow 'b$::{*comm-monoid-add*}
shows $(\sum x \in\# A.\ C\ x + D\ x) = (\sum x \in\# A.\ C\ x) + (\sum x \in\# A.\ D\ x)$
⟨*proof*⟩

lemma *msetsum-union-disjoint*:
assumes $A \#\cap B = \{\#\}$

shows $(\sum La{\in}\#A\ \#\cup\ B.\ f\ La) = (\sum La{\in}\#A.\ f\ La) + (\sum La{\in}\#B.\ f\ La)$
⟨*proof*⟩

1.1.11 Order

Instantiating multiset order as a linear order.

TODO: remove when multiset is of sort ord again

instantiation *multiset* :: (*linorder*) *linorder*
begin

definition *less-multiset* :: *'a*::*linorder multiset* ⇒ *'a multiset* ⇒ *bool* **where**
$M' < M \longleftrightarrow M' \#\subset\# M$

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒*bool* **where**
$(M'::'a\ multiset) \leq M \longleftrightarrow M' \#\subseteq\# M$

instance
⟨*proof*⟩
end
end

1.2 Transitions

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

theory *Wellfounded-More*
imports *Main*

begin

1.2.1 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*⟩

1.2.2 Full Transitions

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

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

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

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

We define output notations only for printing:

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

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*:
$full\ r\ S\ S' \longleftrightarrow ((S = S' \wedge no\text{-}step\ r\ S') \vee full1\ r\ S\ S')$
⟨*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*⟩

1.2.3 Well-Foundedness and Full Transitions

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

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

1.2.4 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-chain*

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*]

1.3 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**
$bounded\ f \longleftrightarrow (\exists b.\ \forall n.\ f\ n \leq b)$

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*⟩

1.4 More List

1.4.1 *upt*

The simplification rules are not very handy, because theorem *upt.simps* (*2*) (i.e. [*?i..<Suc ?j*] = (*if ?i* ≤ *?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* ≤ *?n* ⟹ *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*] ⟷ *A* = [*m ..<m+length A*] ∧ *B* = [*m* + *length A..<n*]

⟨*proof*⟩

A more restrictive version holds:

lemma *B* ≠ [] ⟹ *A* @ *B* = [*m..< n*] ⟷ *A* = [*m ..<m+length A*] ∧ *B* = [*m* + *length A..<n*]
(**is** *?P* ⟹ *?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*⟩

1.4.2 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*⟩

1.4.3 Remove

More lemmas about remove

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

lemma *remove1-mset-single-add*:
a ≠ *b* ⟹ *remove1-mset a* ({#*b*#} + *C*) = {#*b*#} + *remove1-mset a C*
remove1-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* (λ*L*. *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* (λ*x*. ¬*P x*) *xs*
⟨*proof*⟩

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

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 ≡ *case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))

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

Filter

lemma *distinct-filter-eq-if*:
distinct C ⟹ *length* (*filter* (*op* = *L*) *C*) = (*if L* ∈ *set C then 1 else 0*)
⟨*proof*⟩

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*) $\vee$ *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-of $(-L) =$ *atm-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* :: $'a\ literal \Rightarrow 'a\ literal \Rightarrow bool$ **where**
less-literal $L\ M \longleftrightarrow$ *atm-of* $L <$ *atm-of* $M \vee$ *atm-of* $L \leq$ *atm-of* $M \wedge$ *is-neg* $L <$ *is-neg* M

definition *less-eq-literal* :: $'a\ literal \Rightarrow 'a\ literal \Rightarrow bool$ **where**
less-eq-literal $L\ M \longleftrightarrow$ *atm-of* $L <$ *atm-of* $M \vee$ *atm-of* $L \leq$ *atm-of* $M \wedge$ *is-neg* $L \leq$ *is-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*: $L \leq M \Longrightarrow atm\text{-}of\ L \leq atm\text{-}of\ M$
⟨*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** $poss\ AA \equiv \{\#Pos\ A.\ A \in\# AA\#\}$
abbreviation *negs* :: *'a multiset* ⇒ *'a clause* **where** $negs\ AA \equiv \{\#Neg\ A.\ A \in\# AA\#\}$

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

lemma *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*:
atm-of $L \in$ *atm-of* ‘ $I \longleftrightarrow (L \in I \vee -L \in I)$
⟨*proof*⟩

lemma *lits-subseteq-imp-atms-subseteq*: *set-mset* $C \subseteq$ *set-mset* $D \Longrightarrow$ *atms-of* $C \subseteq$ *atms-of* D
⟨*proof*⟩

lemma *atms-empty-iff-empty*[*iff*]: *atms-of* $C = \{\} \longleftrightarrow C = \{\#\}$
⟨*proof*⟩

lemma
atms-of-poss[*simp*]: *atms-of* (*poss* AA) = *set-mset* AA **and**
atms-of-negg[*simp*]: *atms-of* (*negs* AA) = *set-mset* AA
⟨*proof*⟩

lemma *less-eq-Max-atms-of*: $C \neq \{\#\} \Longrightarrow C \mathrel{\#\subseteq\#} D \Longrightarrow$ *Max* (*atms-of* C) $\leq$ *Max* (*atms-of* D)
⟨*proof*⟩

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

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

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

lemma *atms-less-imp-lit-less-pos*: $(\bigwedge B.\ B \in$ *atms-of* $C \Longrightarrow B < A) \Longrightarrow L \in\# C \Longrightarrow L <$ *Pos* A
⟨*proof*⟩

lemma *atms-less-eq-imp-lit-less-eq-neg*: $(\bigwedge B.\ B \in$ *atms-of* $C \Longrightarrow B \leq A) \Longrightarrow L \in\# C \Longrightarrow L \leq$ *Neg* A
⟨*proof*⟩

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* ⇒ *'a literal* ⇒ *bool* (**infix** ⊨*l* *50*) **where**
I ⊨*l* *L* ⟷ (*if is-pos L then* (λ*P*. *P*) *else Not*) (*atm-of L* ∈ *I*)

lemma *true-lit-simps*[*simp*]:
I ⊨*l* *Pos A* ⟷ *A* ∈ *I*
I ⊨*l* *Neg A* ⟷ *A* ∉ *I*
⟨*proof*⟩

lemma *true-lit-iff*[*iff*]: *I* ⊨*l* *L* ⟷ (∃ *A*. *L* = *Pos A* ∧ *A* ∈ *I* ∨ *L* = *Neg A* ∧ *A* ∉ *I*)
⟨*proof*⟩

definition *true-cls* :: *'a interp* ⇒ *'a clause* ⇒ *bool* (**infix** ⊨ *50*) **where**
I ⊨ *C* ⟷ (∃ *L*. *L* ∈# *C* ∧ *I* ⊨*l* *L*)

lemma *true-cls-empty*[*iff*]: ¬ *I* ⊨ {#}
⟨*proof*⟩

lemma *true-cls-singleton*[*iff*]: *I* ⊨ {#*L*#} ⟷ *I* ⊨*l* *L*
⟨*proof*⟩

lemma *true-cls-union*[*iff*]: *I* ⊨ *C* + *D* ⟷ *I* ⊨ *C* ∨ *I* ⊨ *D*
⟨*proof*⟩

lemma *true-cls-mono*: *set-mset C* ⊆ *set-mset D* ⟹ *I* ⊨ *C* ⟹ *I* ⊨ *D*
⟨*proof*⟩

lemma
assumes *I* ⊆ *J*
shows
false-to-true-imp-ex-pos: ¬ *I* ⊨ *C* ⟹ *J* ⊨ *C* ⟹ ∃ *A* ∈ *J*. *Pos A* ∈# *C* **and**
true-to-false-imp-ex-neg: *I* ⊨ *C* ⟹ ¬ *J* ⊨ *C* ⟹ ∃ *A* ∈ *J*. *Neg A* ∈# *C*
⟨*proof*⟩

lemma *true-cls-replicate-mset*[*iff*]: *I* ⊨ *replicate-mset n L* ⟷ *n* ≠ *0* ∧ *I* ⊨*l* *L*
⟨*proof*⟩

lemma *pos-literal-in-imp-true-cls*[*intro*]: *Pos A* ∈# *C* ⟹ *A* ∈ *I* ⟹ *I* ⊨ *C*
⟨*proof*⟩

lemma *neg-literal-notin-imp-true-cls*[*intro*]: *Neg A* ∈# *C* ⟹ *A* ∉ *I* ⟹ *I* ⊨ *C*
⟨*proof*⟩

lemma *pos-neg-in-imp-true*: *Pos A* ∈# *C* ⟹ *Neg A* ∈# *C* ⟹ *I* ⊨ *C*
⟨*proof*⟩

definition *true-clss* :: *'a interp* ⇒ *'a clause set* ⇒ *bool* (**infix** ⊨*s* *50*) **where**
I ⊨*s* *CC* ⟷ (∀ *C* ∈ *CC*. *I* ⊨ *C*)

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

lemma *true-clss-singleton*[*iff*]: *I* ⊨*s* {*C*} ⟷ *I* ⊨ *C*
⟨*proof*⟩

lemma *true-clss-union*[*iff*]: *I* ⊨*s* *CC* ∪ *DD* ⟷ *I* ⊨*s* *CC* ∧ *I* ⊨*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* ⇒ *bool* **where**
consistent-interp I = (∀ *L*. ¬(*L* ∈ *I* ∧ − *L* ∈ *I*))

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

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

lemma *consistent-interp-subset*:
assumes
A ⊆ *B* **and**
consistent-interp B
shows *consistent-interp A*
⟨*proof*⟩

lemma *consistent-interp-change-insert*:
a ∉ *A* ⟹ −*a* ∉ *A* ⟹ *consistent-interp* (*insert* (−*a*) *A*) ⟷ *consistent-interp* (*insert a A*)
⟨*proof*⟩

lemma *consistent-interp-insert-pos*[*simp*]:
a ∉ *A* ⟹ *consistent-interp* (*insert a A*) ⟷ *consistent-interp A* ∧ −*a* ∉ *A*
⟨*proof*⟩

lemma *consistent-interp-insert-not-in*:
consistent-interp A ⟹ *a* ∉ *A* ⟹ −*a* ∉ *A* ⟹ *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* ⇒ *'a set* **where**
atms-of-ms ψs = ⋃ (*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*) = (⋃ *x*∈*b*. *atm-of* ‘ *set x*)
⟨*proof*⟩

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

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

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

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

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

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

lemma *atms-of-ms-insert*[*simp*]:
atms-of-ms (*insert* ψs χs) = *atms-of* $\psi s \cup$ *atms-of-ms* χs
⟨*proof*⟩

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

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

lemma *atms-of-ms-single-set-mset-atns-of*[*simp*]:
atms-of-ms (*single* ‘ *set-mset* B) = *atms-of* B
⟨*proof*⟩

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

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

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

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

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

lemma *in-plus-implies-atm-of-on-atms-of-ms*:
assumes C+{#L#} $\in N$
shows *atm-of* $L \in$ *atms-of-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* $P \in I \vee$ *Neg* $P \in I$) (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

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

Totality

definition *total-over-set* :: *'a interp* ⇒ *'a set* ⇒ *bool* **where**
total-over-set I $S = (\forall l \in S.$ *Pos* $l \in I \vee$ *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* $L \in I \vee$ *Neg* $L \in I$)
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *total-over-m-sum*[*iff*]:
shows *total-over-m I* {*C* + *D*} ⟷ (*total-over-m I* {*C*} ∧ *total-over-m I* {*D*})
⟨*proof*⟩

lemma *total-over-m-union*[*iff*]:
total-over-m I (*A* ∪ *B*) ⟷ (*total-over-m I A* ∧ *total-over-m I B*)
⟨*proof*⟩

lemma *total-over-m-insert*[*iff*]:
total-over-m I (*insert a A*) ⟷ (*total-over-set I* (*atms-of a*) ∧ *total-over-m I A*)
⟨*proof*⟩

lemma *total-over-m-extension*:
fixes *I* :: *'v literal set* **and** *A* :: *'v clauses*
assumes *total*: *total-over-m I A*
shows ∃ *I'*. *total-over-m* (*I* ∪ *I'*) (*A*∪*B*)
∧ (∀ *x*∈*I'*. *atm-of x* ∈ *atms-of-ms B* ∧ *atm-of x* ∉ *atms-of-ms A*)
⟨*proof*⟩

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 ∃ *I'*. *total-over-m* (*I* ∪ *I'*) (*A* ∪ *B*)
∧ (∀ *x*∈*I'*. *atm-of x* ∈ *atms-of-ms B* ∧ *atm-of x* ∉ *atms-of-ms A*) ∧ *consistent-interp* (*I* ∪ *I'*)
⟨*proof*⟩

lemma *total-over-set-atms-of-m*[*simp*]:
total-over-set Ia (*atms-of-s Ia*)
⟨*proof*⟩

lemma *total-over-set-literal-defined*:
assumes {#*A*#} + *D* ∈ *ψs*
and *total-over-set I* (*atms-of-ms ψs*)
shows *A* ∈ *I* ∨ −*A* ∈ *I*
⟨*proof*⟩

lemma *tot-over-m-remove*:
assumes *total-over-m* (*I* ∪ {*L*}) {*ψ*}
and *L*: *L* ∉# *ψ* −*L* ∉# *ψ*
shows *total-over-m I* {*ψ*}
⟨*proof*⟩

lemma *total-union*:
assumes *total-over-m I ψ*
shows *total-over-m* (*I* ∪ *I'*) *ψ*
⟨*proof*⟩

lemma *total-union-2*:
assumes *total-over-m I ψ*
and *total-over-m I' ψ'*
shows *total-over-m* (*I* ∪ *I'*) (*ψ* ∪ *ψ'*)
⟨*proof*⟩

Interpretations

definition *true-cls* :: *'a interp* ⇒ *'a clause* ⇒ *bool* (**infix** ⊨ *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*]: $I \models C + D \longleftrightarrow I \models C \vee I \models D$
  ⟨*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* $n\ L \longleftrightarrow n \neq 0 \wedge I \models l\ L$
  ⟨*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* ⇒ *'a clauses* ⇒ *bool* (**infix** ⊨*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 $I \models s$ ψ
shows $I \cup I' \models s$ ψ
⟨*proof*⟩

lemma *true-clss-union-increase'*[*simp*]:
assumes $I' \models s$ ψ
shows $I \cup I' \models s$ ψ
⟨*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-of* $x \notin$ *atms-of-ms* A
and *atms-of* $L \subseteq$ *atms-of-ms* A
and $I \cup I' \models L$
shows $I \models L$
⟨*proof*⟩

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

Satisfiability

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

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

abbreviation *unsatisfiable* :: *'a clause set* ⇒ *bool* **where**
unsatisfiable CC ≡ ¬ *satisfiable CC*

lemma *satisfiable-decreasing*:
assumes *satisfiable* (*ψ* ∪ *ψ'*)
shows *satisfiable ψ*
⟨*proof*⟩

lemma *satisfiable-def-min*:
satisfiable CC
⟷ (∃ *I*. *I* ⊨*s CC* ∧ *consistent-interp I* ∧ *total-over-m I CC* ∧ *atm-of*`*I* = *atms-of-ms CC*)
(**is** *?sat* ⟷ *?B*)
⟨*proof*⟩

lemma *satisfiable-carac*[*iff*]:
(∃ *I*. *consistent-interp I* ∧ *I* ⊨*s φ*) ⟷ *satisfiable φ* (**is** (∃ *I*. *?Q I*) ⟷ *?S*)
⟨*proof*⟩

lemma *satisfiable-carac'*[*simp*]: *consistent-interp I* ⟹ *I* ⊨*s φ* ⟹ *satisfiable φ*
⟨*proof*⟩

Entailment for Multisets of Clauses

definition *true-cls-mset* :: *'a interp* ⇒ *'a clause multiset* ⇒ *bool* (**infix** ⊨*m* 50) **where**
I ⊨*m CC* ⟷ (∀ *C* ∈# *CC*. *I* ⊨ *C*)

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

lemma *true-cls-mset-singleton*[*iff*]: *I* ⊨*m* {#*C*#} ⟷ *I* ⊨ *C*
⟨*proof*⟩

lemma *true-cls-mset-union*[*iff*]: *I* ⊨*m CC* + *DD* ⟷ *I* ⊨*m CC* ∧ *I* ⊨*m DD*
⟨*proof*⟩

lemma *true-cls-mset-image-mset*[*iff*]: *I* ⊨*m image-mset f A* ⟷ (∀ *x* ∈# *A*. *I* ⊨ *f x*)
⟨*proof*⟩

lemma *true-cls-mset-mono*: *set-mset DD* ⊆ *set-mset CC* ⟹ *I* ⊨*m CC* ⟹ *I* ⊨*m DD*
⟨*proof*⟩

lemma *true-clss-set-mset*[*iff*]: *I* ⊨*s set-mset CC* ⟷ *I* ⊨*m CC*
⟨*proof*⟩

lemma *true-cls-mset-increasing-r*[*simp*]:
I ⊨*m CC* ⟹ *I* ∪ *J* ⊨*m CC*
⟨*proof*⟩

theorem *true-cls-remove-unused*:
assumes *I* ⊨ *ψ*
shows {*v* ∈ *I*. *atm-of v* ∈ *atms-of ψ*} ⊨ *ψ*
⟨*proof*⟩

theorem *true-clss-remove-unused*:
assumes *I* ⊨*s ψ*
shows {*v* ∈ *I*. *atm-of v* ∈ *atms-of-ms ψ*} ⊨*s ψ*

⟨*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-of* $v \notin$ *atms-of* ψ
shows $I \models \psi$
⟨*proof*⟩

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

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

lemma *consistent-interp-disjoint*:
assumes *consI*: *consistent-interp I*
and *disj*: *atms-of-s A* ∩ *atms-of-s I* = {}
and *consA*: *consistent-interp A*
shows *consistent-interp* (*A* ∪ *I*)
⟨*proof*⟩

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

lemma *true-cls-remove-hd-if-notin-vars*:
assumes *insert a M'*⊨ *D*
and *atm-of a* ∉ *atms-of D*
shows *M'* ⊨ *D*
⟨*proof*⟩

lemma *total-over-set-atm-of*:
fixes *I* :: *'v interp* **and** *K* :: *'v set*
shows *total-over-set I K* ⟷ (∀ *l* ∈ *K*. *l* ∈ (*atm-of* ‘ *I*))
⟨*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* (ψ:: *'v clause*) ≡ ∀ *I*. *total-over-set I* (*atms-of* ψ) ⟶ *I* ⊨ ψ

lemma *tautology-Pos-Neg*[*intro*]:
assumes *Pos p* ∈# *A* **and** *Neg p* ∈# *A*
shows *tautology A*
⟨*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*]: ¬*tautology* {#}
⟨*proof*⟩

lemma *tautology-add-single*:
tautology ({#*a*#} + *L*) ⟷ *tautology L* ∨ −*a* ∈# *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* φ
⟨*proof*⟩

lemma *tautology-imp-tautology*:
fixes $\chi\ \chi'$ :: *'v clause*
assumes $\forall I.$ *total-over-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* ⇒ *'a clause* ⇒ *bool* (**infix** ⊨*f 49*) **where**
$\psi \models f\ \chi \longleftrightarrow (\forall I.$ *total-over-m* $I\ (\{\psi\} \cup \{\chi\}) \longrightarrow$ *consistent-interp* $I \longrightarrow I \models \psi \longrightarrow I \models \chi)$

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

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

definition *true-clss-clss* :: *'a clauses* ⇒ *'a clauses* ⇒ *bool* (**infix** ⊨*ps 49*) **where**
$N \models ps\ N' \longleftrightarrow (\forall I.$ *total-over-m* $I\ (N \cup N') \longrightarrow$ *consistent-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*]:
$a \models f\ C \Longrightarrow \mathit{insert}\ a\ A \models p\ C$
⟨*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*]:
$A \models p\ C \Longrightarrow \mathit{insert}\ a\ A \models p\ C$
⟨*proof*⟩

lemma *true-clss-clss-insert-l*[*simp*]:
$A \models ps\ C \Longrightarrow \mathit{insert}\ a\ A \models ps\ C$
⟨*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*]:
$A \models ps\ \mathit{insert}\ L\ Ls \longleftrightarrow (A \models p\ L \wedge A \models ps\ Ls)$
⟨*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*]: $I \models C \mathbin{\#\cup} D \longleftrightarrow I \models C \vee I \models D$
⟨*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 $N \models p\ D \mathbin{\#\cup} C$
⟨*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 $\{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \vee$ *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* ⊆ *atms* ∧ ¬*tautology C* ∧ *distinct-mset C*}

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

lemma *simple-clss-insert*:
assumes *l* ∉ *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 ∈ *simple-clss atms*
shows *atms-of x* ⊆ *atms* ∧ ¬*tautology x* ∧ *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** ¬*tautology* χ
shows χ ∈ *simple-clss* (*atms-of* χ)
⟨*proof*⟩

lemma *simplified-in-simple-clss*:
assumes *distinct-mset-set* ψ **and** ∀ χ ∈ ψ. ¬*tautology* χ
shows ψ ⊆ *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* $L\ I \models e\ x \longleftrightarrow \{L\} \models e\ x \vee I \models e\ x$
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*]: $I \models es\ CC \cup DD \longleftrightarrow I \models es\ CC \wedge I \models es\ DD$
⟨*proof*⟩

lemma *entails-insert*[*iff*]: $I \models es$ *insert* $C\ DD \longleftrightarrow I \models e\ C \wedge I \models es\ DD$
⟨*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** $\models$*sext 49*)
where
$I \models sext\ N \longleftrightarrow (\forall J.\ I \subseteq J \longrightarrow consistent\text{-}interp\ J \longrightarrow total\text{-}over\text{-}m\ J\ N \longrightarrow J \models s\ N)$

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

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

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

lemma *not-consistent-true-clss-ext*:
assumes ¬*consistent-interp I*
shows $I \models sext\ A$
⟨*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 | *FF* | *FVar 'v* | *FNot 'v propo* | *FAnd 'v propo 'v propo* | *FOr 'v propo 'v propo*
  | *FImp 'v propo 'v propo* | *FEq 'v propo 'v propo*

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

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

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

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

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

lemma *propo-induct-arity*[*case-names nullary unary binary*]:
  fixes *φ ψ* :: *'v propo*
  assumes *nullary*: ⋀*φ x*. *φ* = *FF* ∨ *φ* = *FT* ∨ *φ* = *FVar x* ⟹ *P φ*
  and *unary*: ⋀*ψ*. *P ψ* ⟹ *P* (*FNot ψ*)
  and *binary*: ⋀*φ ψ1 ψ2*. *P ψ1* ⟹ *P ψ2* ⟹ *φ* = *FAnd ψ1 ψ2* ∨ *φ* = *FOr ψ1 ψ2* ∨ *φ* = *FImp ψ1 ψ2*
    ∨ *φ* = *FEq ψ1 ψ2* ⟹ *P φ*
  shows *P ψ*
  ⟨*proof*⟩

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*: ⋀*x*. *c* = *CT* ∨ *c* = *CF* ∨ *c* = *CVar x* ⟹ *P*
and *binary*: *c* ∈ *binary-connectives* ⟹ *P*
and *unary*: *c* = *CNot* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *connective-cases-arity-2*[*case-names nullary unary binary*]:
assumes *nullary*: *c* ∈ *nullary-connective* ⟹ *P*
and *unary*: *c* = *CNot* ⟹ *P*
and *binary*: *c* ∈ *binary-connectives* ⟹ *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* ∨ *c* = *CF* ∨ *c* = *CVar v*) ⟹ *wf-conn c* [] |
wf-conn-unary[*simp*]: *c* = *CNot* ⟹ *wf-conn c* [*ψ*] |
wf-conn-binary[*simp*]: *c* ∈ *binary-connectives* ⟹ *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**
⋀*v*. *c* = *CT* ⟹ *P* [] **and**
⋀*v*. *c* = *CF* ⟹ *P* [] **and**
⋀*v*. *c* = *CVar v* ⟹ *P* [] **and**
⋀*ψ*. *c* = *CNot* ⟹ *P* [*ψ*] **and**
⋀*ψ ψ'*. *c* = *COr* ⟹ *P* [*ψ*, *ψ'*] **and**
⋀*ψ ψ'*. *c* = *CAnd* ⟹ *P* [*ψ*, *ψ'*] **and**
⋀*ψ ψ'*. *c* = *CImp* ⟹ *P* [*ψ*, *ψ'*] **and**
⋀*ψ ψ'*. *c* = *CEq* ⟹ *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-conn CT l $\Longrightarrow$ *conn CT l = FT*
wf-conn CF l $\Longrightarrow$ *conn CF l = FF*
wf-conn (CVar x) l $\Longrightarrow$ *conn (CVar x) l = FVar x*
⟨*proof*⟩

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

lemma *wf-conn-list*:
wf-conn c l $\Longrightarrow$ *conn c l = FT* $\longleftrightarrow$ (*c = CT* $\wedge$ *l = []*)
wf-conn c l $\Longrightarrow$ *conn c l = FF* $\longleftrightarrow$ (*c = CF* $\wedge$ *l = []*)
wf-conn c l $\Longrightarrow$ *conn c l = FVar x* $\longleftrightarrow$ (*c = CVar x* $\wedge$ *l = []*)
wf-conn c l $\Longrightarrow$ *conn c l = FAnd a b* $\longleftrightarrow$ (*c = CAnd* $\wedge$ *l = a # b # []*)
wf-conn c l $\Longrightarrow$ *conn c l = FOr a b* $\longleftrightarrow$ (*c = COr* $\wedge$ *l = a # b # []*)
wf-conn c l $\Longrightarrow$ *conn c l = FEq a b* $\longleftrightarrow$ (*c = CEq* $\wedge$ *l = a # b # []*)
wf-conn c l $\Longrightarrow$ *conn c l = FImp a b* $\longleftrightarrow$ (*c = CImp* $\wedge$ *l = a # b # []*)
wf-conn c l $\Longrightarrow$ *conn c l = FNot a* $\longleftrightarrow$ (*c = CNot* $\wedge$ *l = a # []*)
⟨*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-connectives*
shows *length l = 2* $\longleftrightarrow$ *wf-conn c l*
⟨*proof*⟩

lemma *wf-conn-not-list-length*[*iff*]:
fixes *l* :: *'v propo list*
shows *wf-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-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-conn c l* $\longleftrightarrow$ *wf-conn c l'*
⟨*proof*⟩

lemma *wf-conn-no-arity-change-helper*:
length (ξ @ φ # ξ') = *length* (ξ @ φ' # ξ')
⟨*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* ψ
shows *c* = *CNot* **and** *l* = [ψ]
⟨*proof*⟩

lemma *conn-inj*:
fixes *c ca* :: *'v connective* **and** *l* ψs :: *'v propo list*
assumes *corr*: *wf-conn ca l*
and *corr'*: *wf-conn c* ψs
and *eq*: *conn ca l* = *conn c* ψs
shows *ca* = *c* ∧ ψs = *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* ⟹ *wf-conn c l* ⟹ $\varphi \preceq \psi$ ⟹ $\varphi \preceq$ *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* ∈ *binary-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 φ ψ :: *'v propo*
assumes *incl*: $\varphi \preceq \psi$
and *simple*: ψ = *FT* ∨ ψ = *FF* ∨ ψ = *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 $\Longrightarrow \varphi \preceq$ *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 [φ]
wf-conn CT []
wf-conn CF []
wf-conn (*CVar x*) []
wf-conn CAnd [φ, ψ]
wf-conn COr [φ, ψ]
wf-conn CImp [φ, ψ]
wf-conn CEq [φ, ψ]
⟨*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 ξ ξ' :: *'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* = $L \mid R$

We use *nil* instead of ε.

fun *pos* :: *'v propo* ⇒ *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 ⟹ *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* ⇒ *'v propo* ⇒ *'v propo* ⇒ *bool* **where**
path-to-refl[*intro*]: *path-to* [] φ φ |
path-to-l: *c*∈*binary-connectives* ∨ *c* = *CNot* ⟹ *wf-conn c* (φ#*l*) ⟹ *path-to p* φ φ'⟹
path-to (*L*#*p*) (*conn c* (φ#*l*)) φ' |
path-to-r: *c*∈*binary-connectives* ⟹ *wf-conn c* (ψ#φ#[]) ⟹ *path-to p* φ φ' ⟹
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' ⟹ $\varphi' \preceq \varphi$
⟨*proof*⟩

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

fun *replace-at* :: *sign list* ⇒ *'v propo* ⇒ *'v propo* ⇒ *'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* ⇒ *bool*) ⇒ *'v propo* ⇒ *bool* (**infix** ⊨ *50*) **where**
$\mathcal{A}$ ⊨ *FT* = *True* |
$\mathcal{A}$ ⊨ *FF* = *False* |
$\mathcal{A}$ ⊨ *FVar v* = ($\mathcal{A}$ *v*) |
$\mathcal{A}$ ⊨ *FNot* φ = (¬($\mathcal{A}$⊨ φ)) |
$\mathcal{A}$ ⊨ *FAnd* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ∧ $\mathcal{A}$⊨φ_2) |
$\mathcal{A}$ ⊨ *FOr* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ∨ $\mathcal{A}$⊨φ_2) |
$\mathcal{A}$ ⊨ *FImp* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ⟶ $\mathcal{A}$⊨φ_2) |
$\mathcal{A}$ ⊨ *FEq* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ⟷ $\mathcal{A}$ ⊨φ_2)

definition *evalf* (**infix** ⊨*f* *50*) **where**
evalf φ ψ = (∀ *A*. *A* ⊨ φ ⟶ *A* ⊨ ψ)

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 $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists \varphi'.\ \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi')$
⟨*proof*⟩

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

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

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

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

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 φ ψ $\Longrightarrow$ *preserves-un-sat r* $\Longrightarrow$ *propo-rew-step r φ ψ* $\Longrightarrow (\forall A.\ A \models\varphi \longleftrightarrow A\models\psi)$
⟨*proof*⟩

lemma *propo-rew-step-preservers-val'*:
assumes *preserves-un-sat r*
shows *preserves-un-sat* (*propo-rew-step r*)
⟨*proof*⟩

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

lemma *star-consistency-preservation-explicit*:
assumes (*propo-rew-step r*)$\widehat{}**$ *φ ψ* **and** *preserves-un-sat r*
shows $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*proof*⟩

lemma *star-consistency-preservation*:
preserves-un-sat r $\Longrightarrow$ *preserves-un-sat* (*propo-rew-step r*)$\widehat{}**$
⟨*proof*⟩

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' φ $\Longrightarrow$ $\neg(\exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi')$
⟨*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* ⇒ *bool*) ⇒ '*a propo* ⇒ *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 φ $\Longrightarrow$ *test-symb* φ
⟨*proof*⟩

lemma *all-subformula-st-decomp-imp*:
wf-conn c l $\Longrightarrow$ (*test-symb* (*conn c l*) ∧ (∀ $\varphi\in$ *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*) ∧ (∀ $\varphi\in$ *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*) ∧ (∀ $\varphi\in$ *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 φ ψ :: $'v$ *propo*
shows *all-subformula-st test-symb* (*FAnd* φ ψ)
$\longleftrightarrow$ (*test-symb* (*FAnd* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb* (*FOr* φ ψ)
$\longleftrightarrow$ (*test-symb* (*FOr* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb* (*FNot* φ)
$\longleftrightarrow$ (*test-symb* (*FNot* φ) $\wedge$ *all-subformula-st test-symb* φ)
and *all-subformula-st test-symb* (*FEq* φ ψ)
$\longleftrightarrow$ (*test-symb* (*FEq* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb* (*FImp* φ ψ)
$\longleftrightarrow$ (*test-symb* (*FImp* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
⟨*proof*⟩

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 (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ # ξ')) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn* c (ξ @ φ' # ξ')) **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 (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ # ξ'))
$\longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn* c (ξ @ φ' # ξ')) **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 (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ # ξ')) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn* c (ξ @ φ' # ξ')) **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 (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*:
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* φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*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* $\Rightarrow$ *'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* φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*c*:: *'v connective*) *l l'*. *wf-conn c l* $\longrightarrow$ *wf-conn c l'*
$\longrightarrow$ (*test-symb* (*conn c l*) $\longleftrightarrow$ *test-symb* (*conn c l'*)) **and**
full: *full* (*propo-rew-step r*) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *elim-equiv-explicit*: *elim-equiv* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*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* ⇒ *bool* **where**
no-equiv-symb (*FEq* - -) = *False* |
no-equiv-symb - = *True*

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

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

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

lemma *no-equiv-eq*[*simp*]:
fixes φ ψ :: *'v propo*
shows
¬*no-equiv* (*FEq* φ ψ)
no-equiv FT
no-equiv FF
⟨*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) ⟷ *no-equiv* φ
no-equiv (*FAnd* φ ψ) ⟷ (*no-equiv* φ ∧ *no-equiv* ψ)
no-equiv (*FOr* φ ψ) ⟷ (*no-equiv* φ ∧ *no-equiv* ψ)
no-equiv (*FImp* φ ψ) ⟷ (*no-equiv* φ ∧ *no-equiv* ψ)
⟨*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*: ¬ *no-equiv* φ
shows ∃ ψ ψ'. $\psi \preceq \varphi$ ∧ *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 $\psi \Longrightarrow$ *no-equiv* ψ
⟨*proof*⟩

3.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

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

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

lemma *elim-imp-explicit*: *elim-imp* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*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* ≡ *all-subformula-st no-imp-symb*
declare *no-imp-def*[*simp*]

lemma *no-imp-Imp*[*simp*]:
¬*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 ∧ *no-imp* ψ)
no-imp (*FOr* φ ψ) $\longleftrightarrow$ (*no-imp* φ ∧ *no-imp* ψ)
⟨*proof*⟩

Invariant of the *elim-imp* transformation

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

⟨*proof*⟩

lemma *elim-imp-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-imp*) φ ψ **and** *no-equiv* φ
shows *no-equiv* ψ
⟨*proof*⟩

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* ψ ψ'
⟨*proof*⟩

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

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*
⟨*proof*⟩

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* ψs) $\longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall\,\psi{\in}$*set* ψs. $\psi \neq FF \wedge \psi \neq FT))$
⟨*proof*⟩

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

no-T-F-symb (*FImp* φ ψ) $\longleftrightarrow$ ($\forall \chi \in$ *set* [φ, ψ]. $\chi \neq$ *FF* $\wedge$ $\chi \neq$ *FT*)
⟨*proof*⟩

lemma *no-T-F-symb-false*[*simp*]:
fixes *c* :: *'v connective*
shows
¬*no-T-F-symb* (*FT* :: *'v propo*)
¬*no-T-F-symb* (*FF* :: *'v propo*)
⟨*proof*⟩

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

lemma *no-T-F-symb-fnot-imp*:
¬*no-T-F-symb* (*FNot* φ) $\Longrightarrow$ φ = *FT* $\vee$ φ = *FF*
⟨*proof*⟩

lemma *no-T-F-symb-fnot*[*simp*]:
no-T-F-symb (*FNot* φ) $\longleftrightarrow$ ¬(φ = *FT* $\vee$ φ = *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-T-F-symb* φ $\Longrightarrow$ *no-T-F-symb-except-toplevel* φ

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

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

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes φ ψ :: *'v propo*
assumes $\varphi \neq$ *FT* **and** $\varphi \neq$ *FF* **and** $\psi \neq$ *FT* **and** $\psi \neq$ *FF*
and *c*: *c*∈ *binary-connectives*
shows *no-T-F-symb-except-toplevel* (*conn c* [φ, ψ])
⟨*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* ∈ *set l* ∨ *FF* ∈ *set l*
shows ¬*no-T-F-symb-except-toplevel* (*conn c l*)
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-example*[*simp*]:
fixes φ ψ :: *'v propo*
assumes $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
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 $\varphi = FT \vee \varphi = FF$
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 ≡ *all-subformula-st no-T-F-symb-except-toplevel*

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

definition *no-T-F* **where**
no-T-F ≡ *all-subformula-st no-T-F-symb*

lemma *no-T-F-except-top-level-false*:
fixes *l* :: *'v propo list* **and** *c* :: *'v connective*
assumes *wf-conn c l*
and $FT \in$ *set l* $\vee$ $FF \in$ *set l*
shows ¬*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 $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
shows
¬*no-T-F-except-top-level* (*FAnd* φ ψ)
¬*no-T-F-except-top-level* (*FOr* φ ψ)
¬*no-T-F-except-top-level* (*FEq* φ ψ)
¬*no-T-F-except-top-level* (*FImp* φ ψ)
⟨*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

⟨*proof*⟩

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level FF no-T-F-except-top-level FT*
⟨*proof*⟩

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* $\varphi)$
⟨*proof*⟩

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

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* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
⟨*proof*⟩

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

lemma *no-T-F-comp-not*[*simp*]:
fixes φ ψ :: *'v propo*
shows *no-T-F* (*FNot* φ) $\longleftrightarrow$ *no-T-F* φ
⟨*proof*⟩

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

lemma *no-T-F-decomp-not*:
fixes φ :: *'v propo*
assumes φ: *no-T-F* (*FNot* φ)
shows *no-T-F* φ
⟨*proof*⟩

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

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

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

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
⟨*proof*⟩

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* φ ψ) $\longleftrightarrow$ $A \models$ (*FOr* (*FNot* φ) (*FNot* ψ))
$A \models$ *FNot* (*FOr* φ ψ) $\longleftrightarrow$ $A \models$ (*FAnd* (*FNot* φ) (*FNot* ψ))
$A \models$ *FNot* (*FNot* φ) $\longleftrightarrow$ $A \models \varphi$
⟨*proof*⟩

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

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

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

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

lemma *simple-decomp*:
simple $\varphi \longleftrightarrow (\varphi = FT \vee \varphi = FF \vee (\exists x.\ \varphi = FVar\ x))$
⟨*proof*⟩

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

⟨*proof*⟩

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

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* $(FNot\ (FOr\ \varphi\ \psi))$
⟨*proof*⟩

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* $\psi\ \psi'$
⟨*proof*⟩

lemma *simple-not-rew*:
fixes $\varphi :: 'v\ propo$
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* $\psi\ \psi'$
⟨*proof*⟩

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* $(FOr\ (FNot\ \varphi)\ (FNot\ \psi))$
⟨*proof*⟩

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* $(FAnd\ (FNot\ \varphi)\ (FNot\ \psi))$
⟨*proof*⟩

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 $(FNot\ (FNot\ \varphi'))$
⟨*proof*⟩

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
⟨*proof*⟩

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* ⇒ *'v connective* ⇒ *'v propo* ⇒ *'v propo* ⇒ *bool*
for *c c'*:: *'v connective* **where**
push-conn-inside-l[*simp*]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
⟹ *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*]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
⟹ *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 ⟹ ∀ *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*]:
¬*propo-rew-step* (*push-conn-inside c c'*) *FT* ψ ¬*propo-rew-step* (*push-conn-inside c c'*) *FF* ψ
⟨*proof*⟩

inductive *not-c-in-c'-symb*:: *'v connective* ⇒ *'v connective* ⇒ *'v propo* ⇒ *bool* **for** *c c'* **where**
not-c-in-c'-symb-l[*simp*]: *wf-conn c* [*conn c'* [φ, φ'], ψ] ⟹ *wf-conn c'* [φ, φ']
⟹ *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']] ⟹ *wf-conn c'* [φ, φ']
⟹ *not-c-in-c'-symb c c'* (*conn c* [ψ, *conn c'* [φ, φ']])

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

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

⟨*proof*⟩

lemma *c-in-c′-symb-simp′*[*simp*]:
¬*not-c-in-c′-symb c c′ FF*
¬*not-c-in-c′-symb c c′ FT*
¬*not-c-in-c′-symb c c′* (*FVar x*)
¬*not-c-in-c′-symb c c′* (*FNot FF*)
¬*not-c-in-c′-symb c c′* (*FNot FT*)
¬*not-c-in-c′-symb c c′* (*FNot* (*FVar x*))
⟨*proof*⟩

definition *c-in-c′-only* **where**
c-in-c′-only c c′ ≡ *all-subformula-st* (*c-in-c′-symb c c′*)

lemma *c-in-c′-only-simp*[*simp*]:
c-in-c′-only c c′ FF
c-in-c′-only c c′ FT
c-in-c′-only c c′ (*FVar x*)
c-in-c′-only c c′ (*FNot FF*)
c-in-c′-only c c′ (*FNot FT*)
c-in-c′-only c c′ (*FNot* (*FVar x*))
⟨*proof*⟩

lemma *not-c-in-c′-symb-commute*:
not-c-in-c′-symb c c′ ξ ⟹ *wf-conn c* [*φ*, *ψ*] ⟹ *ξ* = *conn c* [*φ*, *ψ*]
⟹ *not-c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])
⟨*proof*⟩

lemma *not-c-in-c′-symb-commute′*:
wf-conn c [*φ*, *ψ*] ⟹ *c-in-c′-symb c c′* (*conn c* [*φ*, *ψ*]) ⟷ *c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])
⟨*proof*⟩

lemma *not-c-in-c′-comm*:
assumes *wf*: *wf-conn c* [*φ*, *ψ*]
shows *c-in-c′-only c c′* (*conn c* [*φ*, *ψ*]) ⟷ *c-in-c′-only c c′* (*conn c* [*ψ*, *φ*]) (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

lemma *not-c-in-c′-simp*[*simp*]:
fixes *φ1 φ2 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′* [*φ1*, *φ2*], *ψ*] ⟹ *wf-conn c′* [*φ1*, *φ2*]
⟹ ¬ *c-in-c′-only c c′* (*conn c* [*conn c′* [*φ1*, *φ2*], *ψ*])
⟨*proof*⟩

lemma *c-in-c′-symb-not*[*simp*]:
fixes *c c′* :: *′v connective* **and** *ψ* :: *′v propo*
shows *c-in-c′-symb c c′* (*FNot ψ*)
⟨*proof*⟩

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

shows $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c' ψ* $\Longrightarrow \exists \psi'$. *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 $\varphi\ \psi$ :: *'v propo*
shows *propo-rew-step* (*push-conn-inside c c'*) *φ ψ* $\Longrightarrow$ *no-T-F φ* $\Longrightarrow$ *no-T-F ψ*
⟨*proof*⟩

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

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

lemma *propo-rew-step-push-conn-inside-simple-not*:
fixes $\varphi\ \varphi'$ :: *'v propo* **and** $\xi\ \xi'$ :: *'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 ψ $\Longrightarrow \psi \neq FT \wedge \psi \neq FF$
⟨*proof*⟩

lemma *push-conn-inside-inv*:
fixes $\varphi\ \psi$ :: *'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 $\varphi\ \psi$ :: *'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**
$c = CAnd \vee c = COr$ **and**
$c' = CAnd \vee c' = COr$
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
$\vee$ ($\exists\ \varphi'.\ \psi$ = *FNot* $\varphi' \wedge$ *simple* φ')
$\vee$ ($\exists\, l.\ \psi$ = *conn* $c\ l \wedge$ *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* $\psi \vee$ ($\exists\ \varphi'.\ \psi$ = *FNot* $\varphi' \wedge$ *simple* φ')
$\vee$ ($\exists\, l.\ \psi$ = *conn* $c\ l \wedge$ *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: $c = CAnd \vee c = COr$ **and** c': $c' = CAnd \vee c' = COr$
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: $c = CAnd \vee c = COr$ **and** c': $c' = CAnd \vee c' = COr$
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: $c = CAnd \vee c = COr$ **and** c': $c' = CAnd \vee c' = COr$
shows *only-c-inside c* $\varphi \Longrightarrow$ *c-in-c′-only c c′* φ
⟨*proof*⟩

lemma *c-in-c′-symb-c-implies-only-c-inside*:
assumes δ: $c = CAnd \vee c = COr$ $c' = CAnd \vee c' = COr$ $c \neq c'$ **and** *wf*: *wf-conn c* [φ, ψ]
and *inv*: *no-equiv* (*conn c l*) *no-imp* (*conn c l*) *simple-not* (*conn c l*)
shows *wf-conn c l* $\Longrightarrow$ *c-in-c′-only c c′* (*conn c l*) $\Longrightarrow$ ($\forall \psi \in$ *set l. only-c-inside c* ψ)
⟨*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* $\psi 1$ $\psi 2$) 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'* [$\varphi 1$, $\varphi 2$]) ⟹ $c' = CAnd \vee c' = COr$ ⟹ *wf-conn c'* [$\varphi 1$, $\varphi 2$]
⟹ $c' = c$
⟨*proof*⟩

lemma *only-c-inside-c-eq-c'*:
only-c-inside c (conn c' [φ1, φ2]) ⟹ c' = CAnd ∨ c' = COr ⟹ wf-conn c' [φ1, φ2] ⟹ c = c'
⟨*proof*⟩

lemma *only-c-inside-imp-grouped-by*:
assumes *c*: *c ≠ CNot* **and** *c'*: *c' = CAnd ∨ c' = COr*
shows *only-c-inside c φ ⟹ grouped-by c φ* (**is** *?O φ ⟹ ?G φ*)
⟨*proof*⟩

lemma *grouped-by-false*:
grouped-by c (conn c' [φ, ψ]) ⟹ c ≠ c' ⟹ wf-conn c' [φ, ψ] ⟹ 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 φ ⟹ super-grouped-by c c' φ* |
connected-is-super-group: *super-grouped-by c c' φ ⟹ super-grouped-by c c' ψ ⟹ wf-conn c [φ, ψ]*
⟹ super-grouped-by c c' (conn c' [φ, ψ])

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 ∨ c = COr* **and** *c'*: *c' = CAnd ∨ c' = COr* **and** *cc'*: *c ≠ c'*
shows *no-equiv φ ⟹ no-imp φ ⟹ simple-not φ ⟹ c-in-c'-only c c' φ*
⟹ super-grouped-by c c' φ
(**is** *?NE φ ⟹ ?NI φ ⟹ ?SN φ ⟹ ?C φ ⟹ ?S φ*)
⟨*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 φ ⟹ no-imp φ ⟹ simple-not φ ⟹ or-in-and-only φ ⟹ is-conj-with-TF φ*
⟨*proof*⟩

definition *is-cnf* **where**
is-cnf φ ≡ is-conj-with-TF φ ∧ 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 $\neg$ *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*: $\neg$ *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 $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *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 $\varphi' \Longrightarrow$ *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 $\varphi' \Longrightarrow$ *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* ⇒ *'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* ⇒ *'a literal multiset set* **where**
mset-of-formula (*FAnd* φ ψ) = *mset-of-formula* φ ∪ *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) ⟷ (∃ *v*. φ = *FVar v* ∨ φ = *FF* ∨ φ = *FT*)
⟨*proof*⟩

lemma *grouped-by-COr-FNot*:
grouped-by COr (*FNot* φ) ⟷ (∃ *v*. φ = *FVar v* ∨ φ = *FF* ∨ φ = *FT*)
⟨*proof*⟩

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

lemma *grouped-by-CAnd-FAnd*:
grouped-by CAnd (*FAnd* φ*1* φ*2*) ⟷ *grouped-by CAnd* φ*1* ∧ *grouped-by CAnd* φ*2*
⟨*proof*⟩

lemma *grouped-by-COr-FOr*:
grouped-by COr (*FOr* φ*1* φ*2*) ⟷ *grouped-by COr* φ*1* ∧ *grouped-by COr* φ*2*
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *grouped-by-COr-mset-of-formula*:
grouped-by COr φ ⟹ *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* ⊨ uses a function assigning *True* or *False*, while *op* ⊨*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 ⟷ *Partial-Clausal-Logic.true-clss* ({*Pos v*|*v*. *A v*} ∪ {*Neg v*|*v*. ¬*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 ⇒ 'v clauses ⇒ 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 $I \models s\ N \longleftrightarrow I \models s\ N'$

⟨*proof*⟩

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

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

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

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

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

4.1.2 Unconstrained Resolution

type-synonym *'v uncon-state* = *'v clauses*
inductive *uncon-res* :: *'v uncon-state* ⇒ *'v uncon-state* ⇒ *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 $\psi \in S'$
⟨*proof*⟩

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

Subsumption

definition *subsumes* :: *'a literal multiset* ⇒ *'a literal multiset* ⇒ *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

⟨*proof*⟩

lemma *subsumes-subsumption*:
assumes *subsumes D χ*
and *C* ⊂# *D* **and** ¬*tautology χ*
shows *subsumes C χ* ⟨*proof*⟩

lemma *subsumes-tautology*:
assumes *subsumes* (*C* + {#*Pos P*#} + {#*Neg P*#}) *χ*
shows *tautology χ*
⟨*proof*⟩

4.1.3 Inference Rule

type-synonym *'v state* = *'v clauses* × (*'v clause* × *'v clause*) *set*
inductive *inference-clause* :: *'v state* ⇒ *'v clause* × (*'v clause* × *'v clause*) *set* ⇒ *bool*
(**infix** ⇒Res *100*) **where**
resolution:
{#*Pos p*#} + *C* ∈ *N* ⟹ {#*Neg p*#} + *D* ∈ *N* ⟹ ({#*Pos p*#} + *C*, {#*Neg p*#} + *D*) ∉ *already-used*
⟹ *inference-clause* (*N*, *already-used*) (*C* + *D*, *already-used* ∪ {({#*Pos p*#} + *C*, {#*Neg p*#} + *D*)}) |
factoring: {#*L*#} + {#*L*#} + *C* ∈ *N* ⟹ *inference-clause* (*N*, *already-used*) (*C* + {#*L*#}, *already-used*)

inductive *inference* :: *'v state* ⇒ *'v state* ⇒ *bool* **where**
inference-step: *inference-clause S* (*clause*, *already-used*)
⟹ *inference S* (*fst S* ∪ {*clause*}, *already-used*)

abbreviation *already-used-inv*
:: *'a literal multiset set* × (*'a literal multiset* × *'a literal multiset*) *set* ⇒ *bool* **where**
already-used-inv state ≡
(∀(*A*, *B*) ∈ *snd state*. ∃ *p*. *Pos p* ∈# *A* ∧ *Neg p* ∈# *B* ∧
((∃ *χ* ∈ *fst state*. *subsumes χ* ((*A* − {#*Pos p*#}) + (*B* − {#*Neg p*#})))
∨ *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* ∪ {*fst S′*}, *snd S′*)
⟨*proof*⟩

lemma *inference-preserves-already-used-inv*:
assumes *inference S S′*
and *already-used-inv S*
shows *already-used-inv S′*
⟨*proof*⟩

lemma *rtranclp-inference-preserves-already-used-inv*:
assumes *rtranclp inference S S′*
and *already-used-inv S*
shows *already-used-inv S′*
⟨*proof*⟩

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* ⇒ *nat* **where**
sem-tree-size Leaf = *0* |
sem-tree-size (*Node - ag ad*) = *1* + *sem-tree-size ag* + *sem-tree-size ad*

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

fun *partial-interps* :: *'v sem-tree* ⇒ *'v interp* ⇒ *'v clauses* ⇒ *bool* **where**
partial-interps Leaf I ψ = (∃ χ. ¬ *I* ⊨ χ ∧ χ ∈ ψ ∧ *total-over-m I* {χ}) |
partial-interps (*Node v ag ad*) *I* ψ ⟷
(*partial-interps ag* (*I* ∪ {*Pos v*}) ψ ∧ *partial-interps ad* (*I*∪ {*Neg v*}) ψ)

lemma *simplify-preserve-partial-leaf*:
simplify N N' ⟹ *partial-interps Leaf I N* ⟹ *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 ∃ *ψ' 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 $\psi' \wedge \{\#\} \in$ *fst* ψ')
⟨*proof*⟩

lemma *inference-completeness*:
  fixes ψ :: *'v ::linorder state*
  assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
  and *finite*: *finite* (*fst* ψ)
  and *snd* $\psi = \{\}$
  shows $\exists \psi'$. (*rtranclp inference* ψ $\psi' \wedge \{\#\} \in$ *fst* ψ')
⟨*proof*⟩

lemma *inference-soundness*:
  fixes ψ :: *'v ::linorder state*
  assumes *rtranclp inference* ψ ψ' **and** $\{\#\} \in$ *fst* ψ'
  shows *unsatisfiable* (*fst* ψ)
  ⟨*proof*⟩

lemma *inference-soundness-and-completeness*:
fixes ψ :: *'v ::linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd* $\psi = \{\}$
shows ($\exists \psi'$. (*inference*** ψ $\psi' \wedge \{\#\} \in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
  ⟨*proof*⟩

4.1.4 Lemma about the simplified state

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

lemma *simplified-count*:
  assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
  shows *count* χ $L \leq 1$
⟨*proof*⟩

lemma *simplified-no-both*:
  assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
  shows $\neg$ ($L \in\# \chi \wedge -L \in\# \chi$)
⟨*proof*⟩

lemma *simplified-not-tautology*:
  assumes *simplified* $\{\psi\}$
  shows $^\sim$*tautology* ψ
⟨*proof*⟩

lemma *simplified-remove*:
  assumes *simplified* $\{\psi\}$
  shows *simplified* $\{\psi - \{\#l\#\}\}$
⟨*proof*⟩

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 ¬*full1 simplify* $\psi\ \psi'$
⟨*proof*⟩

4.1.5 Resolution and Invariants

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

Invariants

lemma *resolution-finite*:
assumes *resolution* $\psi\ \psi'$ **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *rtranclp-resolution-finite*:
assumes *resolution*** $\psi\ \psi'$ **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *resolution-finite-snd*:
assumes *resolution* $\psi\ \psi'$ **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *rtranclp-resolution-finite-snd*:
assumes *resolution*** $\psi\ \psi'$ **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *resolution-always-simplified*:
assumes *resolution* $\psi\ \psi'$

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') ⊆ *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') ⊆ *atms-of-ms* (*fst* ψ)
⟨*proof*⟩

lemma *resolution-include*:
assumes *res*: *resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' ⊆ *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' ⊆ *simple-clss* (*atms-of-ms* (*fst* ψ))
⟨*proof*⟩

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

lemma *already-used-all-simple-vars-incl*:
assumes *vars* ⊆ *vars'*
shows *already-used-all-simple a vars* ⟹ *already-used-all-simple a vars'*
⟨*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*) ⊆ *vars*
shows *already-used-all-simple* (*snd* (*fst S* ∪ {*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*) ⊆ *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 ψ* ≡ *card* (*already-used-top vars* − *ψ*)

lemma *resolution-card-simple-decreasing*:
assumes *res*: *resolution ψ ψ′*
and *a-u-s*: *already-used-all-simple* (*snd ψ*) *vars*
and *finite-v*: *finite vars*
and *finite-fst*: *finite* (*fst ψ*)
and *finite-snd*: *finite* (*snd ψ*)
and *simp*: *simplified* (*fst ψ*)
and *atms-of-ms* (*fst ψ*) ⊆ *vars*
shows *card-simple vars* (*snd ψ′*) < *card-simple vars* (*snd ψ*)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing*:
assumes *tranclp resolution ψ ψ′* **and** *finite-fst*: *finite* (*fst ψ*)
and *already-used-all-simple* (*snd ψ*) *vars*
and *atms-of-ms* (*fst ψ*) ⊆ *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 ψ ψ′*
and *finite-fst*: *finite* (*fst ψ*)
and *empty-snd*: *snd ψ* = {}
and *simplified* (*fst ψ*)
shows *card-simple* (*atms-of-ms* (*fst ψ*)) (*snd ψ′*) < *card-simple* (*atms-of-ms* (*fst ψ*)) (*snd ψ*)
⟨*proof*⟩

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*) ⊆ *vars* ∧ *simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *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*) ⊆ *vars* ∧ ¬*simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
⟨*proof*⟩

lemma *wf-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* ({(*y*:: *'v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ *simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
∪ {(*y*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ ¬ *simplified* (*fst x*) ∧ *finite* (*snd x*) ∧ *finite* (*fst x*)

$\wedge$ *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*)∈*R*) $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *finite-finite-mset-element-of-mset*[*simp*]:
assumes *finite N*
shows *finite* {*f φ L* |*φ L*. *φ* ∈ *N* ∧ *L* ∈# *φ* ∧ *P φ L*}
⟨*proof*⟩

definition *sum-count-ge-2* :: *'a multiset set* ⇒ *nat* (Ξ) **where**
sum-count-ge-2 ≡ *folding.F* (λ*φ*. *op* +(*msetsum* {#*count φ L* |*L* ∈# *φ*. *2* ≤ *count φ L*#})) *0*

interpretation *sum-count-ge-2*:
folding (λ*φ*. *op* +(*msetsum* {#*count φ L* |*L* ∈# *φ*. *2* ≤ *count φ 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*) ⟹ *A* ⊆ *B* ⟹ Ξ *A* ≤ Ξ *B*
⟨*proof*⟩

lemma *simplify-finite-measure-decrease*:
simplify N N' ⟹ *finite N* ⟹ *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*: ¬*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* $\psi = \{\}$
shows $\exists \psi'$. (*resolution*** ψ $\psi' \wedge \{\#\} \in$ *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 $\psi' \wedge \{\#\} \in$ *fst* ψ')) ⟷ *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *simplified-falsity*:
assumes *simp*: *simplified* ψ
and $\{\#\} \in \psi$
shows $\psi = \{\{\#\}\}$
⟨*proof*⟩

lemma *simplify-falsity-in-preserved*:
assumes *simplify* $\chi s\ \chi s'$
and $\{\#\} \in \chi s$
shows $\{\#\} \in \chi s'$
⟨*proof*⟩

lemma *rtranclp-simplify-falsity-in-preserved*:
assumes *simplify*** $\chi s\ \chi s'$
and $\{\#\} \in \chi s$
shows $\{\#\} \in \chi s'$
⟨*proof*⟩

lemma *resolution-falsity-get-falsity-alone*:
assumes *finite* (*fst* ψ)
shows $(\exists\, \psi'.\ (resolution^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow (\exists\, a\text{-}u\text{-}v.\ resolution^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))$
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *resolution-soundness-and-completeness'*:
fixes ψ :: *'v ::linorder state*
assumes
finite: *finite* (*fst* ψ)**and**
snd: *snd* $\psi = \{\}$
shows $(\exists\, a\text{-}u\text{-}v.\ (resolution^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))) \longleftrightarrow unsatisfiable\ (fst\ \psi)$
⟨*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 =
$\{A.\ C \in N \wedge C \neq \{\#\} \wedge Max\ (set\text{-}mset\ C) = Pos\ A \wedge count\ C\ (Pos\ A) \leq 1$
$\wedge \neg (\bigcup D \in \{D.\ D \#\subset\# C\}.\ production\ D) \models h\ C \wedge S\ C = \{\#\}\}$
⟨*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 = $\{A.\ C \in N \wedge C \neq \{\#\} \wedge Max\ (set\text{-}mset\ C) = Pos\ A \wedge count\ C\ (Pos\ A) \leq 1 \wedge \neg$
interp C $\models h\ C \wedge S\ C = \{\#\}\}$
⟨*proof*⟩

abbreviation *productive A* ≡ (*production A* ≠ {})

abbreviation *produces* :: *'a clause* ⇒ *'a* ⇒ *bool* **where**
produces C A ≡ *production C* = {*A*}

lemma *producesD*:
produces C A $\Longrightarrow$ $C \in N \wedge C \neq \{\#\} \wedge Pos\ A = Max\ (set\text{-}mset\ C) \wedge count\ C\ (Pos\ A) \leq 1 \wedge$
$\neg$ *interp C* $\models h\ C \wedge S\ C = \{\#\}$
⟨*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* ⇒ *'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*: $C \mathrel{\#\subseteq\#} D \Longrightarrow interp\ C \subseteq Interp\ D$
⟨*proof*⟩

lemma *less-imp-production-subseteq-interp*: $C \mathrel{\#\subset\#} D \Longrightarrow production\ C \subseteq interp\ D$
⟨*proof*⟩

lemma *less-eq-imp-production-subseteq-Interp*: $C \mathrel{\#\subseteq\#} D \Longrightarrow production\ C \subseteq Interp\ D$
⟨*proof*⟩

lemma *less-imp-Interp-subseteq-interp*: $C \mathrel{\#\subset\#} D \Longrightarrow Interp\ C \subseteq interp\ D$
⟨*proof*⟩

lemma *less-eq-imp-Interp-subseteq-Interp*: $C \mathrel{\#\subseteq\#} D \Longrightarrow Interp\ C \subseteq Interp\ D$
⟨*proof*⟩

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin Interp\ C \Longrightarrow A \in interp\ D \Longrightarrow C \mathrel{\#\subset\#} D$
⟨*proof*⟩

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin interp\ C \Longrightarrow A \in interp\ D \Longrightarrow C \mathrel{\#\subset\#} D$
⟨*proof*⟩

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin Interp\ C \Longrightarrow A \in Interp\ D \Longrightarrow C \mathrel{\#\subset\#} D$
⟨*proof*⟩

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin interp\ C \Longrightarrow A \in Interp\ D \Longrightarrow C \mathrel{\#\subseteq\#} D$
⟨*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*: $Neg\ A \mathrel{\in\#} C$ **and** *d*: $produces\ D\ A$
shows $A \in interp\ C$
⟨*proof*⟩

lemma *neg-notin-Interp-not-produce*: $Neg\ A \mathrel{\in\#} C \Longrightarrow A \notin Interp\ D \Longrightarrow C \mathrel{\#\subseteq\#} D \Longrightarrow \neg\ produces\ D''\ A$
⟨*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 #⊆# D **and**
d-lt-d': D #⊂# D' **and**
c-at-d: *Interp* $D \models h$ C **and**
subs: *interp* $D' \subseteq (\bigcup C \in CC.$ *production* $C)$
shows $(\bigcup C \in CC.$ *production* $C) \models h$ C
⟨*proof*⟩

lemma *true-Interp-imp-interp*: C #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *Interp* $D \models h$ $C \Longrightarrow$ *interp* $D' \models h$ C
⟨*proof*⟩

lemma *true-Interp-imp-Interp*: C #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *Interp* $D \models h$ $C \Longrightarrow$ *Interp* $D' \models h$ C
⟨*proof*⟩

lemma *true-Interp-imp-INTERP*: C #⊆# $D \Longrightarrow$ *Interp* $D \models h$ $C \Longrightarrow$ *INTERP* $\models h$ C
⟨*proof*⟩

lemma *true-interp-imp-general*:
assumes
c-le-d: C #⊆# D **and**
d-lt-d': D #⊂# D' **and**
c-at-d: *interp* $D \models h$ C **and**
subs: *interp* $D' \subseteq (\bigcup C \in CC.$ *production* $C)$
shows $(\bigcup C \in CC.$ *production* $C) \models h$ C
⟨*proof*⟩

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 #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *interp* $D \models h$ $C \Longrightarrow$ *interp* $D' \models h$ C
⟨*proof*⟩

lemma *true-interp-imp-Interp*: C #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *interp* $D \models h$ $C \Longrightarrow$ *Interp* $D' \models h$ C
⟨*proof*⟩

lemma *true-interp-imp-INTERP*: C #⊆# $D \Longrightarrow$ *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#} #⊂# C
shows ¬*produces* C P
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma
assumes D: C+{#*Neg* P#} #⊂# 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* ⇒ *'a clause* ⇒ *'a clause* ⇒ *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* ⇒ *'a clauses* ⇒ *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* ⇒ *'a clauses* ⇒ *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: $abstract\text{-}red\ C\ N$ **and**
    c-lt-d: $C\ \subseteq\#\ D$
  shows $abstract\text{-}red\ D\ N$
⟨*proof*⟩

lemma *true-clss-cls-extended*:
  assumes
    $A \models p\ B$ **and**
    tot: $total\text{-}over\text{-}m\ I\ A$ **and**
    cons: $consistent\text{-}interp\ I$ **and**
    I-A: $I \models s\ A$
  shows $I \models B$
⟨*proof*⟩
lemma

assumes
CP: ¬ *clss-lt N* ({#*C*#} + {#*E*#}) ⊨*p* {#*C*#} + {#*Neg P*#} **and**
clss-lt N ({#*C*#} + {#*E*#}) ⊨*p* {#*E*#} + {#*Pos P*#} ∨ *clss-lt N* ({#*C*#} + {#*E*#}) ⊨*p* {#*C*#} + {#*Neg P*#}
shows *clss-lt N* ({#*C*#} + {#*E*#}) ⊨*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* ∈ *N* ⟶ *B* ∈ *N* ⟶ ¬*redundant A N* ⟶ ¬*redundant B N*
⟶ *superposition-rules A B C* ⟶ *redundant C N* ∨ *C* ∈ *N*)

lemma
assumes
saturated: *saturated N* **and**
finite: *finite N* **and**
empty: {#} ∉ *N*
shows *INTERP N* ⊨*hs N*
⟨*proof*⟩

end

lemma *tautology-is-redundant*:
assumes *tautology C*
shows *abstract-red C N*
⟨*proof*⟩

lemma *subsumed-is-redundant*:
assumes *AB*: *A* ⊂# *B*
and *AN*: *A* ∈ *N*
shows *abstract-red B N*
⟨*proof*⟩

inductive *redundant* :: *'a clause* ⇒ *'a clauses* ⇒ *bool* **where**
subsumption: *A* ∈ *N* ⟹ *A* ⊂# *B* ⟹ *redundant B N*

lemma *redundant-is-redundancy-criterion*:
fixes *A* :: *'a* :: *wellorder clause* **and** *N* :: *'a* :: *wellorder clauses*
assumes *redundant A N*
shows *abstract-red A N*
⟨*proof*⟩

lemma *redundant-mono*:
redundant A N ⟹ *A* ⊆# *B* ⟹ *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-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* ∪ *lits-of l'*
  ⟨*proof*⟩

lemma *finite-lits-of-def*[*simp*]:
  finite (*lits-of-l L*)
  ⟨*proof*⟩

abbreviation *unmark* **where**
unmark ≡ (λ*a*. {#*lit-of a*#})

abbreviation *unmark-s* **where**
unmark-s M ≡ *unmark* ‘ *M*

abbreviation *unmark-l* **where**
unmark-l M ≡ *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 = {} ⟷ *M* = []
⟨*proof*⟩

Entailment

definition *true-annot* :: ('*a*, '*m*) *ann-lits* ⇒ '*a clause* ⇒ *bool* (**infix** ⊨*a* 49) **where**
I ⊨*a C* ⟷ (*lits-of-l I*) ⊨ *C*

definition *true-annots* :: ('*a*, '*m*) *ann-lits* ⇒ '*a clauses* ⇒ *bool* (**infix** ⊨*as* 49) **where**
I ⊨*as CC* ⟷ (∀ *C* ∈ *CC*. *I* ⊨*a C*)

lemma *true-annot-empty-model*[*simp*]:
¬[] ⊨*a* ψ
⟨*proof*⟩

lemma *true-annot-empty*[*simp*]:
¬*I* ⊨*a* {#}
⟨*proof*⟩

lemma *empty-true-annots-def*[*iff*]:
[] ⊨*as* ψ ⟷ ψ = {}
⟨*proof*⟩

lemma *true-annots-empty*[*simp*]:
I ⊨*as* {}
⟨*proof*⟩

lemma *true-annots-single-true-annot*[*iff*]:
I ⊨*as* {*C*} ⟷ *I* ⊨*a C*
⟨*proof*⟩

lemma *true-annot-insert-l*[*simp*]:
M ⊨*a A* ⟹ *L* # *M* ⊨*a A*
⟨*proof*⟩

lemma *true-annots-insert-l* [*simp*]:
M ⊨*as A* ⟹ *L* # *M* ⊨*as A*
⟨*proof*⟩

lemma *true-annots-union*[*iff*]:
M ⊨*as A* ∪ *B* ⟷ (*M* ⊨*as A* ∧ *M* ⊨*as B*)

⟨*proof*⟩

lemma *true-annots-insert*[*iff*]:
M ⊨*as insert a A* ⟷ (*M* ⊨*a a* ∧ *M* ⊨*as A*)
⟨*proof*⟩

Link between ⊨*as* and ⊨*s*:

lemma *true-annots-true-cls*:
I ⊨*as CC* ⟷ *lits-of-l I* ⊨*s CC*
⟨*proof*⟩

lemma *in-lit-of-true-annot*:
a ∈ *lits-of-l M* ⟷ *M* ⊨*a* {#*a*#}
⟨*proof*⟩

lemma *true-annot-lit-of-notin-skip*:
L # *M* ⊨*a A* ⟹ *lit-of L* ∉# *A* ⟹ *M* ⊨*a A*
⟨*proof*⟩

lemma *true-clss-singleton-lit-of-implies-incl*:
I ⊨*s unmark-l MLs* ⟹ *lits-of-l MLs* ⊆ *I*
⟨*proof*⟩

lemma *true-annot-true-clss-cls*:
MLs ⊨*a ψ* ⟹ *set* (*map unmark MLs*) ⊨*p ψ*
⟨*proof*⟩

lemma *true-annots-true-clss-cls*:
MLs ⊨*as ψ* ⟹ *set* (*map unmark MLs*) ⊨*ps ψ*
⟨*proof*⟩

lemma *true-annots-decided-true-cls*[*iff*]:
map Decided M ⊨*as N* ⟷ *set M* ⊨*s N*
⟨*proof*⟩

lemma *true-annot-singleton*[*iff*]: *M* ⊨*a* {#*L*#} ⟷ *L* ∈ *lits-of-l M*
⟨*proof*⟩

lemma *true-annots-true-clss-clss*:
A ⊨*as* Ψ ⟹ *unmark-l A* ⊨*ps* Ψ
⟨*proof*⟩

lemma *true-annot-commute*:
M @ *M′* ⊨*a D* ⟷ *M′* @ *M* ⊨*a D*
⟨*proof*⟩

lemma *true-annots-commute*:
M @ *M′* ⊨*as D* ⟷ *M′* @ *M* ⊨*as D*
⟨*proof*⟩

lemma *true-annot-mono*[*dest*]:
set I ⊆ *set I′* ⟹ *I* ⊨*a N* ⟹ *I′* ⊨*a N*
⟨*proof*⟩

lemma *true-annots-mono*:

set I ⊆ *set I'* ⟹ *I* ⊨*as N* ⟹ *I'* ⊨*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* ⇒ '*a literal* ⇒ *bool*
where
defined-lit I L ⟷ (*Decided L* ∈ *set I*) ∨ (∃ *P*. *Propagated L P* ∈ *set I*)
∨ (*Decided* (−*L*) ∈ *set I*) ∨ (∃ *P*. *Propagated* (−*L*) *P* ∈ *set I*)

abbreviation *undefined-lit* :: ('*a*, '*m*) *ann-lits* ⇒ '*a literal* ⇒ *bool*
where *undefined-lit I L* ≡ ¬*defined-lit I L*

lemma *defined-lit-rev*[*simp*]:
defined-lit (*rev M*) *L* ⟷ *defined-lit M L*
⟨*proof*⟩

lemma *atm-imp-decided-or-proped*:
assumes *x* ∈ *set I*
shows
(*Decided* (− *lit-of x*) ∈ *set I*)
∨ (*Decided* (*lit-of x*) ∈ *set I*)
∨ (∃ *l*. *Propagated* (− *lit-of x*) *l* ∈ *set I*)
∨ (∃ *l*. *Propagated* (*lit-of x*) *l* ∈ *set I*)
⟨*proof*⟩

lemma *literal-is-lit-of-decided*:
assumes *L* = *lit-of x*
shows (*x* = *Decided L*) ∨ (∃ *l'*. *x* = *Propagated L l'*)
⟨*proof*⟩

lemma *true-annot-iff-decided-or-true-lit*:
defined-lit I L ⟷ (*lits-of-l I* ⊨*l L* ∨ *lits-of-l I* ⊨*l* −*L*)
⟨*proof*⟩

lemma *consistent-inter-true-annots-satisfiable*:
consistent-interp (*lits-of-l I*) ⟹ *I* ⊨*as N* ⟹ *satisfiable N*
⟨*proof*⟩

lemma *defined-lit-map*:
defined-lit Ls L ⟷ *atm-of L* ∈ (λ*l*. *atm-of* (*lit-of l*)) ‘ *set Ls*
⟨*proof*⟩

lemma *defined-lit-uminus*[*iff*]:
defined-lit I (−*L*) ⟷ *defined-lit I L*
⟨*proof*⟩

lemma *Decided-Propagated-in-iff-in-lits-of-l*:
defined-lit I L ⟷ (*L* ∈ *lits-of-l I* ∨ −*L* ∈ *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* ∈ *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 = [] ⟷ *False*
  ⟨*proof*⟩

lemma *get-all-ann-decomposition-never-empty-sym*[*iff*]:
  [] = *get-all-ann-decomposition M* ⟷ *False*
  ⟨*proof*⟩

lemma *get-all-ann-decomposition-decomp*:
  hd (*get-all-ann-decomposition S*) = (*a*, *c*) ⟹ *S* = *c* @ *a*
⟨*proof*⟩

lemma *get-all-ann-decomposition-backtrack-split*:
  backtrack-split S = (*M*, *M'*) ⟷ *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*)] ⟹ *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* = [] ∨ (*length a* = *1* ∧ *is-decided* (*hd a*) ∧ *hd a* ∈ *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* = [] ∨ (*is-decided* (*hd a*) ∧ *hd a* ∈ *set M*)
  ⟨*proof*⟩

lemma *get-all-ann-decomposition-snd-not-decided*:
  assumes (*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)
  and *L* ∈ *set b*
  shows ¬*is-decided L*
  ⟨*proof*⟩

lemma *tl-get-all-ann-decomposition-skip-some*:
  assumes *x* ∈ *set* (*tl* (*get-all-ann-decomposition M1*))
  shows *x* ∈ *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 $(x,\ y) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M0\ @\ Decided\ K\ \#\ M1))$
⟨*proof*⟩

lemma *in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend*:
$(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M') \Longrightarrow$
$\exists\, b'.\ (a,\ b'\ @\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M\ @\ M'))$
⟨*proof*⟩

lemma *in-get-all-ann-decomposition-decided-or-empty*:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $a = [] \vee (is\text{-}decided\ (hd\ a))$
⟨*proof*⟩

lemma *get-all-ann-decomposition-remove-undecided-length*:
assumes $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
shows $length\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ M'')) = length\ (get\text{-}all\text{-}ann\text{-}decomposition\ M'')$
⟨*proof*⟩

lemma *get-all-ann-decomposition-not-is-decided-length*:
assumes $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
shows $1 + length\ (get\text{-}all\text{-}ann\text{-}decomposition\ (Propagated\ (-L)\ P\ \#\ M))$
$= length\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ Decided\ L\ \#\ M))$
⟨*proof*⟩

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 $hd\ (get\text{-}all\text{-}ann\text{-}decomposition\ (Propagated\ (-L)\ P\ \#\ M)) = (M0',\ Propagated\ (-L)\ P\ \#\ M0)$
⟨*proof*⟩

lemma *get-all-ann-decomposition-except-last-choice-equal*:
assumes $\forall\, l \in set\ M'.\ \neg is\text{-}decided\ l$
shows $tl\ (get\text{-}all\text{-}ann\text{-}decomposition\ (Propagated\ (-L)\ P\ \#\ M))$
$= tl\ (tl\ (get\text{-}all\text{-}ann\text{-}decomposition\ (M'\ @\ Decided\ L\ \#\ M)))$
⟨*proof*⟩

lemma *get-all-ann-decomposition-hd-hd*:
assumes $get\text{-}all\text{-}ann\text{-}decomposition\ Ls = (M,\ C)\ \#\ (M0,\ M0')\ \#\ l$
shows $tl\ M = M0'\ @\ M0 \wedge is\text{-}decided\ (hd\ M)$
⟨*proof*⟩

lemma *get-all-ann-decomposition-exists-prepend*[*dest*]:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
shows $\exists\, c.\ M = c\ @\ b\ @\ a$
⟨*proof*⟩

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** $set\ a \subseteq set\ M$
⟨*proof*⟩

lemma *get-all-ann-decomposition-exists-prepend'*:
assumes $(a,\ b) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
obtains c **where** $M = c\ @\ b\ @\ a$

⟨*proof*⟩

lemma *union-in-get-all-ann-decomposition-is-subset*:
assumes (*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)
shows *set a* ∪ *set b* ⊆ *set M*
⟨*proof*⟩

lemma *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons*:
∃ *M1 M2*. (*Decided K # M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*c* @ *Decided K # c'*))
⟨*proof*⟩

lemma *fst-get-all-ann-decomposition-prepend-not-decided*:
assumes ∀ *m*∈*set MS*. ¬ *is-decided m*
shows *set* (*map fst* (*get-all-ann-decomposition M*))
= *set* (*map fst* (*get-all-ann-decomposition* (*MS* @ *M*)))
⟨*proof*⟩

Entailment of the Propagated by the Decided Literal

lemma *get-all-ann-decomposition-snd-union*:
set M = ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*)) ∪ {*L* |*L*. *is-decided L* ∧ *L* ∈ *set M*}
(**is** *?M M* = *?U M* ∪ *?Ls M*)
⟨*proof*⟩

definition *all-decomposition-implies* :: *'a literal multiset set*
⇒ (('*a*, '*m*) *ann-lits* × ('*a*, '*m*) *ann-lits*) *list* ⇒ *bool* **where**
all-decomposition-implies N S ⟷ (∀ (*Ls*, *seen*) ∈ *set S*. *unmark-l Ls* ∪ *N* ⊨*ps unmark-l seen*)

lemma *all-decomposition-implies-empty*[*iff*]:
all-decomposition-implies N [] ⟨*proof*⟩

lemma *all-decomposition-implies-single*[*iff*]:
all-decomposition-implies N [(*Ls*, *seen*)] ⟷ *unmark-l Ls* ∪ *N* ⊨*ps unmark-l seen*
⟨*proof*⟩

lemma *all-decomposition-implies-append*[*iff*]:
all-decomposition-implies N (*S* @ *S'*)
⟷ (*all-decomposition-implies N S* ∧ *all-decomposition-implies N S'*)
⟨*proof*⟩

lemma *all-decomposition-implies-cons-pair*[*iff*]:
all-decomposition-implies N ((*Ls*, *seen*) # *S'*)
⟷ (*all-decomposition-implies N* [(*Ls*, *seen*)] ∧ *all-decomposition-implies N S'*)
⟨*proof*⟩

lemma *all-decomposition-implies-cons-single*[*iff*]:
all-decomposition-implies N (*l* # *S'*) ⟷
(*unmark-l* (*fst l*) ∪ *N* ⊨*ps unmark-l* (*snd l*) ∧
all-decomposition-implies N S')
⟨*proof*⟩

lemma *all-decomposition-implies-trail-is-implied*:
assumes *all-decomposition-implies N* (*get-all-ann-decomposition M*)
shows *N* ∪ {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set M*}
⊨*ps unmark* ‘ ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*))
⟨*proof*⟩

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-decomposition-implies N M $\Longrightarrow$ *all-decomposition-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* ⇒ *'v clauses* **where**
CNot $\psi = \{\ \{\#-L\#\} \mid L.\ L \in\# \psi\ \}$

lemma *in-CNot-uminus*[*iff*]:
shows $\{\#L\#\} \in CNot\ \psi \longleftrightarrow -L \in\# \psi$
⟨*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-over-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* ∈ *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* ∈ *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*$) = i \longleftrightarrow a = -i$

⟨*proof*⟩

lemma *true-clss-cls-plus-CNot*:
assumes
CC-L: $A \models p\ CC + \{\#L\#\}$ **and**
CNot-CC: $A \models ps\ CNot\ CC$
shows $A \models p\ \{\#L\#\}$
⟨*proof*⟩

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 $M \models as\ CNot\ A$
⟨*proof*⟩

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:
$A \cup \{B\} \models ps\ \{\{\#\}\} \longleftrightarrow A \models ps\ CNot\ B$
⟨*proof*⟩

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 $M' \models a\ D$
⟨*proof*⟩

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 $M' \models a\ D$
⟨*proof*⟩

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 $M' \models as\ D$ ⟨*proof*⟩

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 $A \models as\ CNot\ B$
⟨*proof*⟩

lemma *CNot-union-mset*[*simp*]:
$CNot\ (A\ \#\cup\ B) = CNot\ A \cup CNot\ B$
⟨*proof*⟩

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*]:
$no\text{-}dup\ (rev\ M) \longleftrightarrow no\text{-}dup\ M$
⟨*proof*⟩

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
assumes $no\text{-}dup\ M$
shows $length\ M = card\ (atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M)$
⟨*proof*⟩

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*) ∈ *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* ⊨*as CNot A*
and −*lit-of L* ∉# *A*
and *no-dup* (*L* # *M*)
shows *M* ⊨*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** ⊨*asm 50*) **where**
I ⊨*asm C* ≡ *I* ⊨*as* (*set-mset C*)

abbreviation *true-clss-clss-m*:: *'v clause multiset* ⇒ *'v clause multiset* ⇒ *bool* (**infix** ⊨*psm 50*)
where
I ⊨*psm C* ≡ *set-mset I* ⊨*ps* (*set-mset C*)

Analog of theorem *true-clss-clss-subsetE*

lemma *true-clss-clssm-subsetE*: *N* ⊨*psm B* $\Longrightarrow$ *A* ⊆# *B* $\Longrightarrow$ *N* ⊨*psm A*
⟨*proof*⟩

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

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

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

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

Other definition using *Union-mset*

lemma *atms-of-mm U* ≡ *set-mset* (⋃# *image-mset* (*image-mset atm-of*) *U*)
⟨*proof*⟩

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

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

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 $\mu_C :: nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat$ **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*:
(∑ *i*=*1*..<*Suc j*. *f i*) = (∑ *i*=*0*..<*j*. *f* (*Suc i*))
⟨*proof*⟩

lemma μ_C*-cons*:
$\mu_C\ s\ b\ (L\ \#\ M) = L * b\ \widehat{}\ (s - 1 - length\ M) + \mu_C\ s\ b\ M$
⟨*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 $\mu_C\ s\ b\ M \geq b\ \widehat{}\ (s - length\ M)$
⟨*proof*⟩

Copy of `~~/src/HOL/ex/NatSum.thy` (but generalized to $0 \leq k$)

lemma *sum-of-powers*: $0 \leq k \Longrightarrow (k - 1) * (\sum i{=}0..{<}n.\ k\widehat{}i) = k\widehat{}n - (1::nat)$
⟨*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 $\mu_C\ s\ b\ M < b\widehat{}s$
⟨*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 $\mu_C\ s\ b\ M < b\ \widehat{}\ s$
⟨*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*). $length\ xs < s \wedge length\ ys < s \wedge$

$(\forall i{<}\ length\ xs.\ xs\ !\ i < b) \wedge (\forall i{<}\ length\ ys.\ ys\ !\ i < b)\}$
⟨*proof*⟩

definition $\nu NOT :: nat \Rightarrow nat \Rightarrow (nat\ list \times nat\ list)\ set$ **where**
$\nu NOT\ s\ base = \{(ys, xs).\ length\ xs < s \wedge length\ ys < s \wedge$
$(\forall i{<}\ length\ xs.\ xs\ !\ i < base) \wedge (\forall i{<}\ length\ ys.\ ys\ !\ i < base) \wedge$
$(ys, xs) \in lenlex\ less\text{-}than\}$

lemma *finite-*νNOT[*simp*]:
finite $(\nu NOT\ s\ base)$
⟨*proof*⟩

lemma *acyclic-*νNOT: *acyclic* $(\nu NOT\ s\ base)$
⟨*proof*⟩

lemma *wf-*νNOT: *wf* $(\nu NOT\ s\ base)$
⟨*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 $M \Longrightarrow -\ lit\text{-}of\ xa = lit\text{-}of\ x \Longrightarrow x \in set\ M \Longrightarrow xa \notin set\ M$
⟨*proof*⟩

lemma *atms-of-ms-single-atm-of*[*simp*]:
atms-of-ms $\{unmark\ L\ |L.\ P\ L\} = atm\text{-}of\ `\ \{lit\text{-}of\ L\ |L.\ P\ L\}$
⟨*proof*⟩

lemma *atms-of-uminus-lit-atm-of-lit-of*:
atms-of $\{\#\ -lit\text{-}of\ x.\ x \in\# A\#\} = atm\text{-}of\ `\ (lit\text{-}of\ `\ (set\text{-}mset\ A))$
⟨*proof*⟩

lemma *atms-of-ms-single-image-atm-of-lit-of*:
atms-of-ms $(unmark\text{-}s\ A) = atm\text{-}of\ `\ (lit\text{-}of\ `\ A)$
⟨*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
$trail :: 'st \Rightarrow ('v, unit)\ ann\text{-}lits$ **and**
$clauses_{NOT} :: 'st \Rightarrow 'v\ clauses$ **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*
begin
abbreviation *state*$_{NOT}$ :: *'st* ⇒ (*'v*, *unit*) *ann-lit list* × *'v clauses* **where**
state$_{NOT}$ *S* ≡ (*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-cls*$_{NOT}$ *remove-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-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* +
assumes
prepend-trail$_{NOT}$:
state$_{NOT}$ (*prepend-trail L st*) = (*L* # *trail st*, *clauses*$_{NOT}$ *st*) **and**
tl-trail$_{NOT}$:
state$_{NOT}$ (*tl-trail st*) = (*tl* (*trail st*), *clauses*$_{NOT}$ *st*) **and**
add-cls$_{NOT}$:
state$_{NOT}$ (*add-cls*$_{NOT}$ *C st*) = (*trail st*, {#*C*#} + *clauses*$_{NOT}$ *st*) **and**
remove-cls$_{NOT}$:
state$_{NOT}$ (*remove-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-tl-trail$_{NOT}$[*simp*]: *trail* (*tl-trail st*) = *tl* (*trail st*) **and**
trail-add-cls$_{NOT}$[*simp*]: *trail* (*add-cls*$_{NOT}$ *C st*) = *trail st* **and**
trail-remove-cls$_{NOT}$[*simp*]: *trail* (*remove-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-add-cls$_{NOT}$[*simp*]:
clauses$_{NOT}$ (*add-cls*$_{NOT}$ *C st*) = {#*C*#} + *clauses*$_{NOT}$ *st* **and**
clauses-remove-cls$_{NOT}$[*simp*]:
clauses$_{NOT}$ (*remove-cls*$_{NOT}$ *C st*) = *removeAll-mset C* (*clauses*$_{NOT}$ *st*)
⟨*proof*⟩

We define the following function doing the backtrack in the trail:

function *reduce-trail-to*$_{NOT}$ :: *'a list* ⇒ *'st* ⇒ *'st* **where**
reduce-trail-to$_{NOT}$ *F S* =
(*if length* (*trail S*) = *length F* ∨ *trail S* = [] *then S else reduce-trail-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* = [] ⟹ *reduce-trail-to$_{NOT}$ F S* = *S* **and**
reduce-trail-to$_{NOT}$-eq-length[*simp*]: *length* (*trail S*) = *length F* ⟹ *reduce-trail-to$_{NOT}$ F S* = *S*
⟨*proof*⟩

lemma *reduce-trail-to$_{NOT}$-length-ne*[*simp*]:
length (*trail S*) ≠ *length F* ⟹ *trail S* ≠ [] ⟹
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*) ≥ *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* ⟹ *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*) ⟹
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* ⟹
trail (*reduce-trail-to$_{NOT}$ F* (*tl-trail S*)) = *F*
⟨*proof*⟩

lemma *reduce-trail-to$_{NOT}$-length*:
length M = length M' ⟹ reduce-trail-to$_{NOT}$ M S = reduce-trail-to$_{NOT}$ M' S
⟨*proof*⟩

abbreviation *trail-weight* **where**
trail-weight S ≡ map ((λl. 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 ⟹ T ∼ U ⟹ S ∼ U
⟨*proof*⟩

lemma
shows
state-eq$_{NOT}$-trail: *S ∼ T ⟹ trail S = trail T* **and**
state-eq$_{NOT}$-clauses: *S ∼ T ⟹ 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-cls$_{NOT}$ remove-cls$_{NOT}$ propagate-conds +
decide-ops trail clauses$_{NOT}$ prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$ +
backjumping-ops trail clauses$_{NOT}$ prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$ backjump-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
bj-can-jump:
⋀*S C F' K F L*.
inv S ⟹
no-dup (*trail S*) ⟹
trail S = *F'* @ *Decided K* # *F* ⟹
C ∈# *clauses$_{NOT}$ S* ⟹
trail S ⊨*as CNot C* ⟹
undefined-lit F L ⟹
atm-of L ∈ *atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ‘ (*lits-of-l* (*F'* @ *Decided K* # *F*)) ⟹
clauses$_{NOT}$ S ⊨*pm C'* + {#*L*#} ⟹
F ⊨*as CNot C'* ⟹
¬*no-step backjump S*
begin

We cannot add a like condition *atms-of C'* ⊆ *atms-of-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-of L* ∈ *atm-of* ‘ *lits-of-l* (*F'* @ *Decided K* # *F*) is important, otherwise you are not sure that you can backtrack.

Definition

We define dpll with backjumping:

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

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

$\bigwedge L\ T.\ undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
$\Longrightarrow T \sim prepend\text{-}trail\ (Decided\ L)\ S$
$\Longrightarrow P\ S\ T$ **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 $dpll\text{-}bj\ S\ T$ **and** $inv\ S$
shows $clauses_{NOT}\ S = clauses_{NOT}\ T$
⟨*proof*⟩

No duplicates in the trail **lemma** *dpll-bj-no-dup*:
assumes $dpll\text{-}bj\ S\ T$ **and** $inv\ S$
and $no\text{-}dup\ (trail\ S)$
shows $no\text{-}dup\ (trail\ T)$
⟨*proof*⟩

Valuations **lemma** *dpll-bj-sat-iff*:
assumes $dpll\text{-}bj\ S\ T$ **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
$dpll\text{-}bj\ S\ T$ **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
$dpll\text{-}bj\ S\ T$ **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 $dpll\text{-}bj\ S\ T$**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-bj*** *S T* **and**
inv S
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 *rtranclp-dpll-bj-all-decomposition-implies-inv*:
assumes
*dpll-bj*** *S T* **and**
inv S
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-dpll-bj-inv-incl-dpll-bj-inv-trancl*:
{(*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*}
⊆ {(*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** *?A* ⊆ *?B*$^{+}$)
⟨*proof*⟩

lemma *wf-tranclp-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*}
⟨*proof*⟩

lemma *dpll-bj-sat-ext-iff*:
dpll-bj S T ⟹ *inv S* ⟹ *I*⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I*⊨*sextm clauses*$_{NOT}$ *T*
⟨*proof*⟩

lemma *rtranclp-dpll-bj-sat-ext-iff*:
*dpll-bj*** *S T* ⟹ *inv S* ⟹ *I*⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I*⊨*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-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
∨ (*trail T* ⊨*asm clauses*$_{NOT}$ *S* ∧ *satisfiable* (*set-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* ⇒ (*'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
    forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin
inductive *forget$_{NOT}$* :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget$_{NOT}$:
  removeAll-mset C(*clauses$_{NOT}$ S*) ⊨*pm C* ⟹
  forget-cond C S ⟹
  C ∈# *clauses$_{NOT}$ S* ⟹
  T ∼ *remove-cls$_{NOT}$ C S* ⟹
  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* ⇒ (*'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**
    learn-cond forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin
inductive *learn-and-forget$_{NOT}$* :: *'st* ⇒ *'st* ⇒ *bool*
where
lf-learn: *learn S T* ⟹ *learn-and-forget$_{NOT}$ S T* |
lf-forget: *forget$_{NOT}$ S T* ⟹ *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* ⇒ (*'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

inductive *cdcl*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
c-dpll-bj: *dpll-bj S S'* ⟹ *cdcl*$_{NOT}$ *S S'* |
c-learn: *learn S S'* ⟹ *cdcl*$_{NOT}$ *S S'* |
c-forget$_{NOT}$: *forget*$_{NOT}$ *S S'* ⟹ *cdcl*$_{NOT}$ *S S'*

lemma *cdcl*$_{NOT}$*-all-induct*[*consumes 1*, *case-names dpll-bj learn forget*$_{NOT}$]:
fixes *S T* :: *'st*
assumes *cdcl*$_{NOT}$ *S T* **and**
dpll: ⋀*T*. *dpll-bj S T* ⟹ *P S T* **and**
learning:
⋀*C T*. *clauses*$_{NOT}$ *S* ⊨*pm C* ⟹
atms-of C ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ' (*lits-of-l* (*trail S*)) ⟹
T ∼ *add-cls*$_{NOT}$ *C S* ⟹
P S T **and**
forgetting: ⋀*C T*. *removeAll-mset C* (*clauses*$_{NOT}$ *S*) ⊨*pm C* ⟹
C ∈# *clauses*$_{NOT}$ *S* ⟹
T ∼ *remove-cls*$_{NOT}$ *C S* ⟹
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-of* ‘(*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *A* ∧ *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *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*⟙*sextm clauses*$_{NOT}$ *S* ⟷ *I*⟙*sextm clauses*$_{NOT}$ *T*
⟨*proof*⟩

definition *cdcl*$_{NOT}$*-NOT-all-inv* **where**
cdcl$_{NOT}$*-NOT-all-inv A S* ⟷ (*finite A* ∧ *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*))

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 ≡ *learn S T* ∨ *forget*$_{NOT}$ *S T*

lemma *rtranclp-learn-or-forget-cdcl*$_{NOT}$:
learn-or-forget** *S T* ⟹ *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** *?μ U* < *?μ S*)
⟨*proof*⟩

lemma *infinite-cdcl*$_{NOT}$*-exists-learn-and-forget-infinite-chain*:
assumes
⋀*i*. *cdcl*$_{NOT}$ (*f i*) (*f*(*Suc i*)) **and**
inv: *cdcl*$_{NOT}$*-NOT-all-inv A* (*f 0*)
shows ∃ *j*. ∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*))
⟨*proof*⟩

lemma *wf-cdcl*$_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: ⋀*f j*. ¬ (∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*)))
shows *wf* {(*T*, *S*). *cdcl*$_{NOT}$ *S T* ∧ *cdcl*$_{NOT}$*-NOT-all-inv A S*}

(**is** *wf* {(*T*, *S*). $cdcl_{NOT}$ *S T* ∧ *?inv S*})
⟨*proof*⟩

lemma *inv-and-tranclp-cdcl-*$_{NOT}$*-tranclp-cdcl*$_{NOT}$*-and-inv*:
$cdcl_{NOT}{}^{++}$ *S T* ∧ $cdcl_{NOT}$*-NOT-all-inv A S* ⟷ (λ*S T*. $cdcl_{NOT}$ *S T* ∧ $cdcl_{NOT}$*-NOT-all-inv A S*)$^{++}$ *S T*
(**is** *?A* ∧ *?I* ⟷ *?B*)
⟨*proof*⟩

lemma *wf-tranclp-cdcl*$_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: ⋀*f j*. ¬ (∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*)))
shows *wf* {(*T*, *S*). $cdcl_{NOT}{}^{++}$ *S T* ∧ $cdcl_{NOT}$*-NOT-all-inv A S*}
⟨*proof*⟩

lemma $cdcl_{NOT}$*-final-state*:
assumes
n-s: *no-step* $cdcl_{NOT}$ *S* **and**
inv: $cdcl_{NOT}$*-NOT-all-inv A S* **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ *S*))
∨ (*trail S* ⊨*asm* $clauses_{NOT}$ *S* ∧ *satisfiable* (*set-mset* ($clauses_{NOT}$ *S*)))
⟨*proof*⟩

lemma *full-cdcl*$_{NOT}$*-final-state*:
assumes
full: *full* $cdcl_{NOT}$ *S T* **and**
inv: $cdcl_{NOT}$*-NOT-all-inv A S* **and**
n-d: *no-dup* (*trail S*) **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ *T*))
∨ (*trail T* ⊨*asm* $clauses_{NOT}$ *T* ∧ *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-cls*$_{NOT}$ *remove-cls*$_{NOT}$ +
conflict-driven-clause-learning trail $clauses_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds
λ*C S*. *distinct-mset C* ∧ ¬*tautology C* ∧ *learn-restrictions C S* ∧
(∃ *F K d F' C' L*. *trail S* = *F'* @ *Decided K* # *F* ∧ *C* = *C'* + {#*L*#} ∧ *F* ⊨*as CNot C'*
∧ *C'* + {#*L*#} ∉# $clauses_{NOT}$ *S*)
λ*C S*. ¬(∃ *F' F K d L*. *trail S* = *F'* @ *Decided K* # *F* ∧ *F* ⊨*as CNot* (*remove1-mset L C*))
∧ *forget-restrictions C S*

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-restrictions forget-restrictions :: *'v clause* ⇒ *'st* ⇒ *bool*
begin

lemma *cdcl*$_{NOT}$*-learn-all-induct*[*consumes 1*, *case-names dpll-bj learn forget*$_{NOT}$]:
fixes *S T* :: *'st*
assumes *cdcl*$_{NOT}$ *S T* **and**
dpll: ⋀*T*. *dpll-bj S T* ⟹ *P S T* **and**
learning:
⋀*C F K F' C' L T*. *clauses*$_{NOT}$ *S* ⊨*pm C* ⟹
atms-of C ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*)) ⟹
distinct-mset C ⟹
¬ *tautology C* ⟹
learn-restrictions C S ⟹
trail S = *F'* @ *Decided K* # *F* ⟹
C = *C'* + {#*L*#} ⟹
F ⊨*as CNot C'* ⟹
C' + {#*L*#} ∉# *clauses*$_{NOT}$ *S* ⟹
T ∼ *add-cls*$_{NOT}$ *C S* ⟹
P S T **and**
forgetting: ⋀*C T*. *removeAll-mset C* (*clauses*$_{NOT}$ *S*) ⊨*pm C* ⟹
C ∈# *clauses*$_{NOT}$ *S* ⟹
¬(∃*F' F K L*. *trail S* = *F'* @ *Decided K* # *F* ∧ *F* ⊨*as CNot* (*C* − {#*L*#})) ⟹
T ∼ *remove-cls*$_{NOT}$ *C S* ⟹
forget-restrictions C S ⟹
P S T
shows *P S T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-inv*:
cdcl$_{NOT}$** *S T* ⟹ *inv S* ⟹ *inv T*
⟨*proof*⟩

lemma *learn-always-simple-clauses*:
assumes
learn: *learn S T* **and**
n-d: *no-dup* (*trail S*)
shows *set-mset* (*clauses*$_{NOT}$ *T* − *clauses*$_{NOT}$ *S*)
⊆ *simple-clss* (*atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ *lits-of-l* (*trail S*))
⟨*proof*⟩

definition *conflicting-bj-clss S* ≡
{*C*+{#*L*#} |*C L*. *C*+{#*L*#} ∈# *clauses*$_{NOT}$ *S* ∧ *distinct-mset* (*C*+{#*L*#})
∧ ¬*tautology* (*C*+{#*L*#})
∧ (∃*F' K F*. *trail S* = *F'* @ *Decided K* # *F* ∧ *F* ⊨*as CNot C*)}

lemma *conflicting-bj-clss-remove-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*:
$\{a.\ (a = b \vee Q\ a) \wedge S\ a\} = (\textit{if}\ S\ b\ \textit{then}\ \{b\}\ \textit{else}\ \{\}) \cup \{a.\ Q\ a \wedge S\ a\}$
⟨*proof*⟩

lemma *set-insert-neq*:
$A \neq \textit{insert}\ a\ A \longleftrightarrow a \notin A$
⟨*proof*⟩

lemma *learn-μ_L-decrease*:
assumes *learnST*: *learn* $S\ T$ **and** *n-d*: *no-dup* (*trail* S) **and**
A: *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq A$ **and**
fin-A: *finite* A
shows $(\mu_L\ (\textit{card}\ A)\ T,\ \mu_L\ (\textit{card}\ A)\ S) \in$ *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* (*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**
$\mu_{CDCL}\ A\ T \equiv ((2{+}\textit{card}\ (\textit{atms-of-ms}\ A))\ \widehat{}\ (1{+}\textit{card}\ (\textit{atms-of-ms}\ A))$
$\quad - \mu_C\ (1{+}\textit{card}\ (\textit{atms-of-ms}\ A))\ (2{+}\textit{card}\ (\textit{atms-of-ms}\ A))\ (\textit{trail-weight}\ T),$
$\quad \textit{conflicting-bj-clss-yet}\ (\textit{card}\ (\textit{atms-of-ms}\ A))\ T,\ \textit{card}\ (\textit{set-mset}\ (\textit{clauses}_{NOT}\ T)))$
lemma *cdcl$_{NOT}$-decreasing-measure*:
assumes
cdcl$_{NOT}$ $S\ T$ **and**
inv: *inv* S **and**
atm-clss: *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *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 $(\mu_{CDCL}\ A\ T,\ \mu_{CDCL}\ A\ S)$
$\quad \in$ *less-than* $<*lex*>$ (*less-than* $<*lex*>$ *less-than*)
⟨*proof*⟩

lemma *wf-cdcl$_{NOT}$-restricted-learning*:
assumes *finite* A
shows *wf* $\{(T,\ S).$
$\quad (\textit{atms-of-mm}\ (\textit{clauses}_{NOT}\ S) \subseteq \textit{atms-of-ms}\ A \wedge \textit{atm-of}\ ‘\ \textit{lits-of-l}\ (\textit{trail}\ S) \subseteq \textit{atms-of-ms}\ A$
$\quad \wedge\ \textit{no-dup}\ (\textit{trail}\ S)$
$\quad \wedge\ \textit{inv}\ S)$
$\quad \wedge\ \textit{cdcl}_{NOT}\ S\ T\ \}$
⟨*proof*⟩

definition μ_C' :: *'v clause set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_C'\ A\ T \equiv \mu_C\ (1{+}\textit{card}\ (\textit{atms-of-ms}\ A))\ (2{+}\textit{card}\ (\textit{atms-of-ms}\ A))\ (\textit{trail-weight}\ T)$

definition μ_{CDCL}' :: *'v clause set* ⇒ *'st* ⇒ *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-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* {*C*|*C*. *C* ∈# *clauses*$_{NOT}$ *T* ∧ (*tautology C* ∨ ¬*distinct-mset C*)}
≤ *card* {*C*|*C*. *C*∈# *clauses*$_{NOT}$ *S* ∧ (*tautology C* ∨ ¬*distinct-mset C*)} + *3* ^ (*card A*)
(**is** *card ?T* ≤ *card ?S* + -)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-card-simple-clauses-bound*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
NA: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
MA: *atm-of* ' (*lits-of-l* (*trail S*)) ⊆ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* (*set-mset* (*clauses*$_{NOT}$ *T*))
≤ *card* {*C*. *C* ∈# *clauses*$_{NOT}$ *S* ∧ (*tautology C* ∨ ¬*distinct-mset C*)} + *3* ^ (*card A*)
(**is** *card ?T* ≤ *card ?S* + -)
⟨*proof*⟩

definition *μ*$_{CDCL}$*'-bound* :: *'v clause set* ⇒ *'st* ⇒ *nat* **where**
μ$_{CDCL}$*'-bound A S* =
((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))) ∗ (*1* + *3* ^ *card* (*atms-of-ms A*)) ∗ *2*
+ *2*∗*3* ^ (*card* (*atms-of-ms A*))
+ *card* {*C*. *C* ∈# *clauses*$_{NOT}$ *S* ∧ (*tautology C* ∨ ¬*distinct-mset C*)} + *3* ^ (*card* (*atms-of-ms A*))

lemma *μ*$_{CDCL}$*'-bound-reduce-trail-to*$_{NOT}$[*simp*]:
μ$_{CDCL}$*'-bound A* (*reduce-trail-to*$_{NOT}$ *M S*) = *μ*$_{CDCL}$*'-bound A S*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-μ*$_{CDCL}$*'-bound-reduce-trail-to*$_{NOT}$:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite* (*atms-of-ms A*) **and**
U: *U* ∼ *reduce-trail-to*$_{NOT}$ *M T*
shows *μ*$_{CDCL}$*' A U* ≤ *μ*$_{CDCL}$*'-bound A S*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-μ*$_{CDCL}$*'-bound*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite* (*atms-of-ms A*)
shows *μ*$_{CDCL}$*' A T* ≤ *μ*$_{CDCL}$*'-bound A S*
⟨*proof*⟩

lemma *rtranclp-μ*$_{CDCL}$*'-bound-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 $T \leq \mu_{CDCL}$'-*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 :: $'st \Rightarrow 'st \Rightarrow bool$ **and**
    $cdcl_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ +
  fixes
    f :: $nat \Rightarrow nat$ **and**
    bound-inv :: $'bound \Rightarrow 'st \Rightarrow bool$ **and**
    μ :: $'bound \Rightarrow 'st \Rightarrow nat$ **and**
    $cdcl_{NOT}$-*inv* :: $'st \Rightarrow bool$ **and**
    μ-*bound* :: $'bound \Rightarrow 'st \Rightarrow nat$
  assumes
    f: *unbounded* f **and**
    f-ge-1:$\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \neq 0$ **and**
    bound-inv: $\bigwedge A\ S\ T.\ cdcl_{NOT}$-*inv* $S \Longrightarrow$ *bound-inv* $A\ S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow$ *bound-inv* $A\ T$ **and**
    $cdcl_{NOT}$-*measure*: $\bigwedge A\ S\ T.\ cdcl_{NOT}$-*inv* $S \Longrightarrow$ *bound-inv* $A\ S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow \mu\ A\ T < \mu\ A\ S$ **and**
    measure-bound2: $\bigwedge A\ T\ U.\ cdcl_{NOT}$-*inv* $T \Longrightarrow$ *bound-inv* $A\ T \Longrightarrow cdcl_{NOT}{}^{**}\ T\ U$
      $\Longrightarrow \mu\ A\ U \leq \mu$-*bound* $A\ T$ **and**
    measure-bound4: $\bigwedge A\ T\ U.\ cdcl_{NOT}$-*inv* $T \Longrightarrow$ *bound-inv* $A\ T \Longrightarrow cdcl_{NOT}{}^{**}\ T\ U$
      $\Longrightarrow \mu$-*bound* $A\ U \leq \mu$-*bound* $A\ T$ **and**
    $cdcl_{NOT}$-*restart-inv*: $\bigwedge A\ U\ V.\ cdcl_{NOT}$-*inv* $U \Longrightarrow$ *restart* $U\ V \Longrightarrow$ *bound-inv* $A\ U \Longrightarrow$ *bound-inv* $A\ V$
    and
    exists-bound: $\bigwedge R\ S.\ cdcl_{NOT}$-*inv* $R \Longrightarrow$ *restart* $R\ S \Longrightarrow \exists A.$ *bound-inv* $A\ S$ **and**
    $cdcl_{NOT}$-*inv*: $\bigwedge S\ T.\ cdcl_{NOT}$-*inv* $S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow cdcl_{NOT}$-*inv* T **and**
    $cdcl_{NOT}$-*inv-restart*: $\bigwedge S\ T.\ cdcl_{NOT}$-*inv* $S \Longrightarrow$ *restart* $S\ T \Longrightarrow cdcl_{NOT}$-*inv* T
begin

lemma $cdcl_{NOT}$-$cdcl_{NOT}$-*inv*:
  assumes
    $(cdcl_{NOT}{}^{\frown\frown} n)\ S\ T$ **and**
    $cdcl_{NOT}$-*inv* S
  shows $cdcl_{NOT}$-*inv* T
  ⟨*proof*⟩

lemma $cdcl_{NOT}$-*bound-inv*:
  assumes

$(cdcl_{NOT}\ ^{\frown}n)\ S\ T$ **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
$(cdcl_{NOT}\ ^{\frown}(Suc\ n))\ S\ T$ **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 $\neg(cdcl_{NOT}\ ^{\frown}\ m)\ S\ T$
⟨*proof*⟩

- $f\ n < m$ ensures that at least one step has been done.

inductive $cdcl_{NOT}$*-restart* **where**
restart-step: $(cdcl_{NOT}\ ^{\frown}m)\ S\ T \Longrightarrow m \geq f\ n \Longrightarrow restart\ T\ U$
$\Longrightarrow cdcl_{NOT}\text{-}restart\ (S, n)\ (U, Suc\ n)$ |
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-cls$_{NOT}$ remove-cls$_{NOT}$ +
  forget-ops trail clauses$_{NOT}$ prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$ forget-cond +
  propagate-ops trail clauses$_{NOT}$ prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$ 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**
    propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
    forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool* +
  fixes *backjump-l-cond* :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *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*
  ⟹ *no-dup* (*trail S*)
  ⟹ *T* ∼ *prepend-trail* (*Propagated L* ()) (*reduce-trail-to$_{NOT}$ F* (*add-cls$_{NOT}$ C'' S*))
  ⟹ *C* ∈# *clauses$_{NOT}$ S*
  ⟹ *trail S* ⊨*as CNot C*
  ⟹ *undefined-lit F L*
  ⟹ *atm-of L* ∈ *atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
  ⟹ *clauses$_{NOT}$ S* ⊨*pm C'* + {#*L*#}
  ⟹ *C''* = *C'* + {#*L*#}
  ⟹ *F* ⊨*as CNot C'*
  ⟹ *backjump-l-cond C C' L S T*
  ⟹ *backjump-l S T*

Avoid (meaningless) simplification in the theorem generated by *inductive-cases*:

declare *reduce-trail-to$_{NOT}$-length-ne*[*simp del*] *Set.Un-iff*[*simp del*] *Set.insert-iff*[*simp del*]
inductive-cases *backjump-lE*: *backjump-l S T*
thm *backjump-lE*
declare *reduce-trail-to$_{NOT}$-length-ne*[*simp*] *Set.Un-iff*[*simp*] *Set.insert-iff*[*simp*]

inductive *cdcl$_{NOT}$-merged-bj-learn* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
cdcl$_{NOT}$-merged-bj-learn-decide$_{NOT}$: *decide$_{NOT}$ S S'* ⟹ *cdcl$_{NOT}$-merged-bj-learn S S'* |
cdcl$_{NOT}$-merged-bj-learn-propagate$_{NOT}$: *propagate$_{NOT}$ S S'* ⟹ *cdcl$_{NOT}$-merged-bj-learn S S'* |
cdcl$_{NOT}$-merged-bj-learn-backjump-l: *backjump-l S S'* ⟹ *cdcl$_{NOT}$-merged-bj-learn S S'* |
cdcl$_{NOT}$-merged-bj-learn-forget$_{NOT}$: *forget$_{NOT}$ S S'* ⟹ *cdcl$_{NOT}$-merged-bj-learn S S'*

lemma *cdcl$_{NOT}$-merged-bj-learn-no-dup-inv*:
  cdcl$_{NOT}$-merged-bj-learn S T ⟹ *no-dup* (*trail S*) ⟹ *no-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-cls$_{NOT}$ remove-cls$_{NOT}$
    propagate-conds forget-cond
    λ*C C' L' S T*. *backjump-l-cond C C' L' S T*
    ∧ *distinct-mset* (*C'* + {#*L'*#}) ∧ ¬*tautology* (*C'* + {#*L'*#})

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**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool* **and**
backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* +
fixes
inv :: *'st* ⇒ *bool*
assumes
bj-merge-can-jump:
⋀*S C F' K F L*.
inv S
⟹ *trail S* = *F'* @ *Decided K* # *F*
⟹ *C* ∈# *clauses*$_{NOT}$ *S*
⟹ *trail S* ⊨*as CNot C*
⟹ *undefined-lit F L*
⟹ *atm-of L* ∈ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*F'* @ *Decided K* # *F*))
⟹ *clauses*$_{NOT}$ *S* ⊨*pm C'* + {#*L*#}
⟹ *F* ⊨*as CNot C'*
⟹ ¬*no-step backjump-l S* **and**
cdcl-merged-inv: ⋀*S T*. *cdcl*$_{NOT}$*-merged-bj-learn S T* ⟹ *inv S* ⟹ *inv T*
begin

abbreviation *backjump-conds* :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool*
where
backjump-conds ≡ λ*C C' L' S T*. *distinct-mset* (*C'* + {#*L'*#}) ∧ ¬*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-cls*$_{NOT}$ *remove-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-cls*$_{NOT}$ *remove-cls*$_{NOT}$
propagate-conds forget-cond backjump-l-cond inv
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**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool* **and**
backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
inv :: *'st* ⇒ *bool*
begin

sublocale *conflict-driven-clause-learning-ops 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*⟩
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 μ_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 μ_{CDCL}'*-merged* :: *'v clause set* ⇒ *'st* ⇒ *nat* **where**
μ_{CDCL}'*-merged A T* ≡
((*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) − $\mu_C'\ A\ T$) ∗ *2* + *card* (*set-mset* (*clauses*$_{NOT}$ *T*))

lemma *cdcl*$_{NOT}$*-decreasing-measure'*:
assumes
cdcl$_{NOT}$*-merged-bj-learn S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-trail: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows μ_{CDCL}'*-merged A T* < μ_{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*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses$_{NOT}$ S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses$_{NOT}$ S*))
∨ (*trail S* ⊨*asm clauses$_{NOT}$ S* ∧ *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*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses$_{NOT}$ S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses$_{NOT}$ T*))
∨ (*trail T* ⊨*asm clauses$_{NOT}$ T* ∧ *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* ⇒ (*'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-restrictions forget-restrictions :: *'v clause* ⇒ *'st* ⇒ *bool*
+
fixes *f* :: *nat* ⇒ *nat*
assumes
unbounded: *unbounded f* **and** *f-ge-1*: ⋀*n*. *n* ≥ *1* ⟹ *f n* ≥ *1* **and**
inv-restart:⋀*S T*. *inv S* ⟹ *T* ∼ *reduce-trail-to$_{NOT}$* ([]::*'a list*) *S* ⟹ *inv T*
begin

lemma *bound-inv-inv*:
assumes

inv S **and**
n-d: *no-dup* (*trail S*) **and**
atms-clss-S-A: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail-S-A:*atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
finite A **and**
cdcl$_{NOT}$: *cdcl*$_{NOT}$ *S T*
shows
atms-of-mm (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A* **and**
finite A
⟨*proof*⟩

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 $I \models sextm\ clauses_{NOT}\ (fst\ S) \longleftrightarrow I \models sextm\ clauses_{NOT}\ (fst\ T)$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$-*restart-sat-ext-iff*:
fixes S T :: $'st \times nat$
assumes
st: $cdcl_{NOT}$-*restart*** S T **and**
n-d: *no-dup* (*trail* (*fst* S)) **and**
inv: *inv* (*fst* S)
shows $I \models sextm\ clauses_{NOT}\ (fst\ S) \longleftrightarrow I \models sextm\ clauses_{NOT}\ (fst\ T)$
⟨*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)) $\models sextm\ clauses_{NOT}\ S \wedge$ *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-cls*$_{NOT}$ *remove-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* ⇒ (*'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**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
inv :: *'st* ⇒ *bool* **and**
forget-conds :: *'v clause* ⇒ *'st* ⇒ *bool* **and**
backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool*
+
fixes *f* :: *nat* ⇒ *nat*
assumes
unbounded: *unbounded f* **and** *f-ge-1*: ⋀*n*. *n* ≥ *1* ⟹ *f n* ≥ *1* **and**
inv-restart:⋀*S T*. *inv S* ⟹ *T* ∼ *reduce-trail-to*$_{NOT}$ [] *S* ⟹ *inv T*
begin

definition *not-simplified-cls* :: *'b literal multiset multiset* ⇒ *'b literal multiset multiset*
where
not-simplified-cls A ≡ {#*C* ∈# *A*. *C* ∉ *simple-clss* (*atms-of-mm A*)#}

lemma *not-simplified-cls-tautology-distinct-mset*:
not-simplified-cls A = {#*C* ∈# *A*. *tautology C* ∨ ¬*distinct-mset C*#}
⟨*proof*⟩

lemma *simple-clss-or-not-simplified-cls*:
assumes *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
x ∈# *clauses*$_{NOT}$ *S* **and** *finite A*
shows *x* ∈ *simple-clss* (*atms-of-ms A*) ∨ *x* ∈# *not-simplified-cls* (*clauses*$_{NOT}$ *S*)
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound*:
assumes
cdcl$_{NOT}$*-merged-bj-learn S T* **and**
inv: *inv S* **and**
atms-clss: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A[*simp*]: *finite A*
shows *set-mset* (*clauses*$_{NOT}$ *T*) ⊆ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ *S*))
∪ *simple-clss* (*atms-of-ms A*)
⟨*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*) ⊆# *not-simplified-cls* (*clauses*$_{NOT}$ *S*)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl*$_{NOT}$*-merged-bj-learn*** *S T*
shows *not-simplified-cls* (*clauses*$_{NOT}$ *T*) ⊆# *not-simplified-cls* (*clauses*$_{NOT}$ *S*)
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound*:
assumes

$cdcl_{NOT}$-merged-bj-learn** 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**
$finite$[$simp$]: $finite$ A
shows $set\text{-}mset$ $(clauses_{NOT}$ $T) \subseteq$ $set\text{-}mset$ $(not\text{-}simplified\text{-}cls$ $(clauses_{NOT}$ $S))$
$\cup$ $simple\text{-}clss$ $(atms\text{-}of\text{-}ms$ $A)$
⟨*proof*⟩

abbreviation μ_{CDCL}'-$bound$ **where**
μ_{CDCL}'-$bound$ A $T \equiv ((2{+}card$ $(atms\text{-}of\text{-}ms$ $A))$ $\widehat{}$ $(1{+}card$ $(atms\text{-}of\text{-}ms$ $A))) * 2$
$+$ $card$ $(set\text{-}mset$ $(not\text{-}simplified\text{-}cls(clauses_{NOT}$ $T)))$
$+$ 3 $\widehat{}$ $card$ $(atms\text{-}of\text{-}ms$ $A)$

lemma $rtranclp\text{-}cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}clauses\text{-}bound\text{-}card$:
assumes
$cdcl_{NOT}$-merged-bj-learn** 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**
$finite$: $finite$ A
shows μ_{CDCL}'-$merged$ A $T \leq \mu_{CDCL}'$-$bound$ A S
⟨*proof*⟩

sublocale $cdcl_{NOT}$-$increasing\text{-}restarts\text{-}ops$ λS T. $T \sim reduce\text{-}trail\text{-}to_{NOT}$ ([]::'a $list$) S
$cdcl_{NOT}$-merged-bj-learn f
λA S. $atms\text{-}of\text{-}mm$ $(clauses_{NOT}$ $S) \subseteq$ $atms\text{-}of\text{-}ms$ A
$\wedge$ $atm\text{-}of$ ' $lits\text{-}of\text{-}l$ $(trail$ $S) \subseteq$ $atms\text{-}of\text{-}ms$ $A \wedge$ $finite$ A
μ_{CDCL}'-$merged$
λS. inv $S \wedge$ $no\text{-}dup$ $(trail$ $S)$
μ_{CDCL}'-$bound$
⟨*proof*⟩

lemma $cdcl_{NOT}\text{-}restart\text{-}\mu_{CDCL}'\text{-}merged\text{-}le\text{-}\mu_{CDCL}'\text{-}bound$:
assumes
$cdcl_{NOT}$-restart T V
inv $(fst$ $T)$ **and**
$no\text{-}dup$ $(trail$ $(fst$ $T))$ **and**
$atms\text{-}of\text{-}mm$ $(clauses_{NOT}$ $(fst$ $T)) \subseteq$ $atms\text{-}of\text{-}ms$ A **and**
$atm\text{-}of$ ' $lits\text{-}of\text{-}l$ $(trail$ $(fst$ $T)) \subseteq$ $atms\text{-}of\text{-}ms$ A **and**
$finite$ A
shows μ_{CDCL}'-$merged$ A $(fst$ $V) \leq \mu_{CDCL}'$-$bound$ A $(fst$ $T)$
⟨*proof*⟩

lemma $cdcl_{NOT}\text{-}restart\text{-}\mu_{CDCL}'\text{-}bound\text{-}le\text{-}\mu_{CDCL}'\text{-}bound$:
assumes
$cdcl_{NOT}$-restart T V **and**
$no\text{-}dup$ $(trail$ $(fst$ $T))$ **and**
inv $(fst$ $T)$ **and**
fin: $finite$ A
shows μ_{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*)) $\widehat{}$ (*1*+*card* (*atms-of-ms A*))) $*$ *2*
    + *card* (*set-mset* (*not-simplified-cls*(*clauses$_{NOT}$ T*)))
    + *3* $\widehat{}$ *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*⊨*sextm clauses$_{NOT}$* (*fst S*) ⟷ *I* ⊨*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*⊨*sextm clauses$_{NOT}$* (*fst S*) ⟷ *I* ⊨*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*)))

∨ *lits-of-l* (*trail* (*fst T*)) ⊨*sextm* *clauses*$_{NOT}$ (*fst S*) ∧
satisfiable (*set-mset* (*clauses*$_{NOT}$ (*fst S*)))
⟨*proof*⟩

corollary *full-cdcl*$_{NOT}$*-restart-normal-form-init-state*:
assumes
init-state: *trail S* = [] *clauses*$_{NOT}$ *S* = *N* **and**
full: *full cdcl*$_{NOT}$*-restart* (*S*, *0*) *T* **and**
inv: *inv S*
shows *unsatisfiable* (*set-mset N*)
∨ *lits-of-l* (*trail* (*fst T*)) ⊨*sextm N* ∧ *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* × *'v clauses*
⇒ (*'v*, *unit*) *ann-lits* × *'v clauses* ⇒ *bool* **where**
backtrack-split (*fst S*) = (*M'*, *L* # *M*) ⟹ *is-decided L* ⟹ *D* ∈# *snd S*
⟹ *fst S* ⊨*as CNot D* ⟹ *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* ∘ *fst*) (*M*, *N*) **and**
decomp: *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)
shows
∃ *C F' K F L l C'*.
M = *F'* @ *Decided K* # *F* ∧
M' = *Propagated L l* # *F* ∧ *N* = *N'* ∧ *C* ∈# *N* ∧ *F'* @ *Decided K* # *F* ⊨*as CNot C* ∧
undefined-lit F L ∧ *atm-of L* ∈ *atms-of-mm N* ∪ *atm-of* ‘ *lits-of-l* (*F'* @ *Decided K* # *F*) ∧
N ⊨*pm C'* + {#*L*#} ∧ *F* ⊨*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* ∘ *fst*) *S* **and**
decomp: *all-decomposition-implies-m* (*snd S*) (*get-all-ann-decomposition* (*fst S*))
shows
∃ *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 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)
⟨*proof*⟩

sublocale *backjumping-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- - - 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 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 ∧ 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 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 ∧ 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-lits*, *N'*::'*v clauses*), ([], *N*))|*M' N' N*.
dpll-bj^{++} ([], *N*) (*M'*, *N'*) ∧ *atms-of-mm N* ⊆ *atms-of-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*) ∨ (*M'* ⊨*asm N* ∧ *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'* ⊨*asm N* ⟷ *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* ∧ *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* ∧ *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* ⟷ *dpll-bj S T*
⟨*proof*⟩

Another proof of termination:

lemma *wf* {(*T*, *S*). *dpll-bj S T* ∧ *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* ⇒ *nat*
assumes *unbounded*: *unbounded* f **and** *f-ge-1*:⋀*n*. *n*≥ *1* ⟹ $f\ n \geq 1$
begin
sublocale $cdcl_{NOT}$*-increasing-restarts*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*) λ(*M*, *N*). (*tl M*, *N*)
λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*) *f* λ(-, *N*) *S*. *S* = ([], *N*)
λ*A* (*M*, *N*). *atms-of-mm N* ⊆ *atms-of-ms A* ∧ *atm-of* ' *lits-of-l M* ⊆ *atms-of-ms A* ∧ *finite A*
∧ *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)
λ*A T*. (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*) *dpll-bj*
λ(*M*, *N*). *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)
λ*A* -. (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-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* × *'v clauses*

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

inductive $dpll_W$ :: *'v* $dpll_W$*-state* ⇒ *'v* $dpll_W$*-state* ⇒ *bool* **where**
propagate: *C* + {#*L*#} ∈# *clauses S* ⟹ *trail S* ⊨*as CNot C* ⟹ *undefined-lit* (*trail S*) *L*
⟹ $dpll_W$ *S* (*Propagated L* () # *trail S*, *clauses S*) |
decided: *undefined-lit* (*trail S*) *L* ⟹ *atm-of L* ∈ *atms-of-mm* (*clauses S*)
⟹ $dpll_W$ *S* (*Decided L* # *trail S*, *clauses S*) |
backtrack: *backtrack-split* (*trail S*) = (*M'*, *L* # *M*) ⟹ *is-decided L* ⟹ *D* ∈# *clauses S*
⟹ *trail S* ⊨*as CNot D* ⟹ $dpll_W$ *S* (*Propagated* (− (*lit-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 ((λa. {#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 (λa. {#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 $dpll_W\text{-}all\text{-}inv\ S \equiv$
$(all\text{-}decomposition\text{-}implies\text{-}m\ (clauses\ S)\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S))$
$\wedge\ atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S)\ \subseteq\ atms\text{-}of\text{-}mm\ (clauses\ S)$
$\wedge\ consistent\text{-}interp\ (lits\text{-}of\text{-}l\ (trail\ S))$
$\wedge\ no\text{-}dup\ (trail\ S))$

lemma $dpll_W\text{-}all\text{-}inv\text{-}dest[dest]$:
assumes $dpll_W\text{-}all\text{-}inv\ S$
shows $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses\ S)\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S))$
and $atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S)\ \subseteq\ atms\text{-}of\text{-}mm\ (clauses\ S)$
and $consistent\text{-}interp\ (lits\text{-}of\text{-}l\ (trail\ S))\ \wedge\ no\text{-}dup\ (trail\ S)$
⟨*proof*⟩

lemma $rtranclp\text{-}dpll_W\text{-}all\text{-}inv$:
assumes $rtranclp\ dpll_W\ S\ S'$
and $dpll_W\text{-}all\text{-}inv\ S$
shows $dpll_W\text{-}all\text{-}inv\ S'$
⟨*proof*⟩

lemma $dpll_W\text{-}all\text{-}inv$:
assumes $dpll_W\ S\ S'$
and $dpll_W\text{-}all\text{-}inv\ S$
shows $dpll_W\text{-}all\text{-}inv\ S'$
⟨*proof*⟩

lemma $rtranclp\text{-}dpll_W\text{-}inv\text{-}starting\text{-}from\text{-}0$:
assumes $rtranclp\ dpll_W\ S\ S'$
and inv: $trail\ S = []$
shows $dpll_W\text{-}all\text{-}inv\ S'$
⟨*proof*⟩

lemma $dpll_W\text{-}can\text{-}do\text{-}step$:
assumes $consistent\text{-}interp\ (set\ M)$
and $distinct\ M$
and $atm\text{-}of\ `\ (set\ M) \subseteq atms\text{-}of\text{-}mm\ N$
shows $rtranclp\ dpll_W\ ([], N)\ (map\ Decided\ M, N)$
⟨*proof*⟩

definition $conclusive\text{-}dpll_W\text{-}state\ (S::\ 'v\ dpll_W\text{-}state) \longleftrightarrow$
$(trail\ S \models asm\ clauses\ S \vee ((\forall L \in set\ (trail\ S).\ \neg is\text{-}decided\ L)$
$\wedge\ (\exists C \in\#\ clauses\ S.\ trail\ S \models as\ CNot\ C)))$

theorem 2.8.6 page 74 of Weidenbach's book

lemma $dpll_W\text{-}strong\text{-}completeness$:
assumes $set\ M \models sm\ N$
and $consistent\text{-}interp\ (set\ M)$
and $distinct\ M$
and $atm\text{-}of\ `\ (set\ M) \subseteq atms\text{-}of\text{-}mm\ N$
shows $dpll_W^{**}\ ([], N)\ (map\ Decided\ M, N)$
and $conclusive\text{-}dpll_W\text{-}state\ (map\ Decided\ \ M, N)$
⟨*proof*⟩

theorem 2.8.5 page 73 of Weidenbach's book

lemma $dpll_W\text{-}sound$:
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-dpll$_W$-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$_W$-card-decrease*:
assumes *dpll*: $dpll_W\ S\ S'$ **and** $length\ (trail\ S') \leq card\ vars$
and $length\ (trail\ S) \leq card\ vars$
shows $(dpll_W\text{-}mes\ (trail\ S')\ (card\ vars),\ dpll_W\text{-}mes\ (trail\ S)\ (card\ vars))$
$\in lexn\ \{(a,\ b).\ a < b\}\ (card\ vars)$
⟨*proof*⟩

theorem 2.8.7 page 74 of Weidenbach's book

lemma *dpll$_W$-card-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$_W$-wf*:
$wf\ \{(S',\ S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}$
⟨*proof*⟩

lemma *dpll$_W$-tranclp-star-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$_W$-wf-tranclp*: $wf\ \{(S',\ S).\ dpll_W\text{-}all\text{-}inv\ S \wedge {dpll_W}^{++}\ S\ S'\}$
⟨*proof*⟩

lemma *dpll$_W$-wf-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* ⊨*asm N* ⟷ *satisfiable* (*set-mset N*) (**is** *?A* ⟷ *?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* ⟷ *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* ⊨*asm N* ⟷ *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* ⇒ *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* ≡ *Max* (*set-mset M*)

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

definition *get-maximum-level* :: (*'a*, *'b*) *ann-lits* ⇒ *'a literal multiset* ⇒ *nat*
  where
get-maximum-level M D = *MMax* ({#*0*#} + *image-mset* (*get-level M*) *D*)

lemma *get-maximum-level-ge-get-level*:
  L ∈# *D* ⟹ *get-maximum-level M D* ≥ *get-level M L*
  ⟨*proof*⟩

lemma *get-maximum-level-empty*[*simp*]:
  get-maximum-level M {#} = *0*
  ⟨*proof*⟩

lemma *get-maximum-level-exists-lit-of-max-level*:
  D ≠ {#} ⟹ ∃ *L*∈# *D*. *get-level M L* = *get-maximum-level M D*
  ⟨*proof*⟩

lemma *get-maximum-level-empty-list*[*simp*]:
  get-maximum-level [] *D* = *0*
  ⟨*proof*⟩

lemma *get-maximum-level-single*[*simp*]:
  get-maximum-level M {#*L*#} = *get-level M L*
  ⟨*proof*⟩

lemma *get-maximum-level-plus*:
  get-maximum-level M (*D* + *D'*) = *max* (*get-maximum-level M D*) (*get-maximum-level M D'*)
  ⟨*proof*⟩

lemma *get-maximum-level-exists-lit*:
  assumes *n*: *n* > *0*
  and *max*: *get-maximum-level M D* = *n*
  shows ∃ *L* ∈#*D*. *get-level M L* = *n*
⟨*proof*⟩

lemma *get-maximum-level-skip-first*[*simp*]:
  assumes *atm-of L* ∉ *atms-of D*
  shows *get-maximum-level* (*Propagated L C* # *M*) *D* = *get-maximum-level M D*
  ⟨*proof*⟩

lemma *get-maximum-level-skip-beginning*:
  assumes *DH*: ∀ *x* ∈ *atms-of D*. *x* ∉ *atm-of* ‘ *lits-of-l c*
  shows *get-maximum-level* (*c* @ *H*) *D* = *get-maximum-level H D*
⟨*proof*⟩

lemma *get-maximum-level-D-single-propagated*:
  get-maximum-level [*Propagated x21 x22*] *D* = *0*
  ⟨*proof*⟩

lemma *get-maximum-level-skip-un-decided-not-present*:
  assumes
    ∀ *L*∈#*D*. *atm-of L* ∉ *atm-of* ‘ *lits-of-l M* **and**
    ∀ *m*∈*set M*. ¬ *is-decided m*
  shows *get-maximum-level* (*M* @ *aa*) *D* = *get-maximum-level aa D*

⟨*proof*⟩

lemma *get-maximum-level-union-mset*:
get-maximum-level M (*A* #∪ *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 ≥ *get-level M L*
⟨*proof*⟩

lemma *count-decided-ge-get-maximum-level*:
count-decided M ≥ *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*:
(λ*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* ⟷ (∃ *c K c*′. *M* = *c* @ *Decided K* # *c*′ ∧ *get-level M K* = *Suc i*)
(**is** *?A* ⟷ *?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* ⇒ (*'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* +
assumes
cons-trail:
⋀*S'*. *state st* = (*M*, *S'*) ⟹
state (*cons-trail L st*) = (*L* # *M*, *S'*) **and**

tl-trail:
⋀*S'*. *state st* = (*M*, *S'*) ⟹ *state* (*tl-trail st*) = (*tl M*, *S'*) **and**

remove-cls:
⋀*S'*. *state st* = (*M*, *N*, *U*, *S'*) ⟹
state (*remove-cls C st*) =
(*M*, *removeAll-mset C N*, *removeAll-mset C U*, *S'*) **and**

add-learned-cls:
⋀*S'*. *state st* = (*M*, *N*, *U*, *S'*) ⟹
state (*add-learned-cls C st*) = (*M*, *N*, {#*C*#} + *U*, *S'*) **and**

update-backtrack-lvl:
⋀*S'*. *state st* = (*M*, *N*, *U*, *k*, *S'*) ⟹
state (*update-backtrack-lvl k' st*) = (*M*, *N*, *U*, *k'*, *S'*) **and**

update-conflicting:
state st = (*M*, *N*, *U*, *k*, *D*) ⟹
state (*update-conflicting E st*) = (*M*, *N*, *U*, *k*, *E*) **and**

init-state:
state (*init-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 ∼ *T* ⟷ *state S* = *state T*

lemma *state-eq-ref*[*simp*, *intro*]:
S ∼ *S*
⟨*proof*⟩

lemma *state-eq-sym*:
S ∼ *T* ⟷ *T* ∼ *S*
⟨*proof*⟩

lemma *state-eq-trans*:
S ∼ *T* ⟹ *T* ∼ *U* ⟹ *S* ∼ *U*
⟨*proof*⟩

lemma
shows
state-eq-trail: *S* ∼ *T* ⟹ *trail S* = *trail T* **and**
state-eq-init-clss: *S* ∼ *T* ⟹ *init-clss S* = *init-clss T* **and**
state-eq-learned-clss: *S* ∼ *T* ⟹ *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 $op \sim$ 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*[*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* ⟷ *conflicting S* = *None*
⟨*proof*⟩

lemma *trail-eq-reduce-trail-to-eq*:
trail S = *trail T* ⟹ *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
⟨*proof*⟩

lemma *reduce-trail-to-state-eq*$_{NOT}$*-compatible*:
assumes *ST*: *S* ∼ *T*
shows *reduce-trail-to F S* ∼ *reduce-trail-to F T*
⟨*proof*⟩

lemma *reduce-trail-to-trail-tl-trail-decomp*[*simp*]:
trail S = *F'* @ *Decided K* # *F* ⟹ (*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'* ⟹ *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*) ≥ *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*) ∈ *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* ⟷ *conflicting S* = *None*
  ⟨*proof*⟩

lemma *conflicting-add-learned-cls-conflicting*[*simp*]:
  conflicting (*add-learned-cls C S*) = *None* ⟷ *conflicting S* = *None*
  ⟨*proof*⟩

lemma *conflicting-update-backtracl-lvl*[*simp*]:
  conflicting (*update-backtrack-lvl k S*) = *None* ⟷ *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-driven-clause-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* ⇒ (*'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

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 ⟹
   conflicting S = Some E ⟹
   −L ∉# E ⟹
   E ≠ {#} ⟹
   T ∼ tl-trail S ⟹
   skip S T

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

get-maximum-level (*Propagated L* (*C* + {#*L*#}) # *M*) *D* = *k* ∨ *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 ≠* [] ⟹
  hd-trail S = Propagated L E ⟹
  L ∈# E ⟹
  conflicting S = Some D′ ⟹
  −L ∈# D′ ⟹
  get-maximum-level (*trail S*) ((*remove1-mset* (*−L*) *D′*)) = *backtrack-lvl S ⟹*
  T ∼ update-conflicting (*Some* (*resolve-cls L D′ E*))
    (*tl-trail S*) ⟹
  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*) ⟹
  ¬M ⊨asm clauses S ⟹
  U′ ⊆# U ⟹
  state T = ([], *N*, *U′*, *0*, *None*) ⟹
  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 ⟹
  C ∈# learned-clss S ⟹
  ¬(*trail S*) *⊨asm clauses S ⟹*
  C ∉ set (*get-all-mark-of-propagated* (*trail S*)) ⟹
  C ∉# init-clss S ⟹
  T ∼ remove-cls C S ⟹
  forget S T

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

inductive $cdcl_W$*-rf* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
restart: *restart S T ⟹* $cdcl_W$*-rf S T* |
forget: *forget S T ⟹* $cdcl_W$*-rf S T*

inductive $cdcl_W$*-bj* :: *'st* ⇒ *'st* ⇒ *bool* **where**
skip: *skip S S′ ⟹* $cdcl_W$*-bj S S′* |
resolve: *resolve S S′ ⟹* $cdcl_W$*-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* $\Longrightarrow$
¬(*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*: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and** *bt*: $backtrack\ S\ T$
shows $backtrack\text{-}lvl\ T < backtrack\text{-}lvl\ S$
⟨*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: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
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** ¬*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-clss S = init-clss S′*
⟨*proof*⟩

lemma *cdcl$_W$-init-clss*:
assumes
cdcl$_W$ S T **and**
inv: *cdcl$_W$-M-level-inv S*
shows *init-clss S = init-clss T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-init-clss*:
cdcl$_W$** S T ⟹ *cdcl$_W$-M-level-inv S* ⟹ *init-clss S = init-clss T*
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-init-clss*:
cdcl$_W$$^{++}$ S T ⟹ *cdcl$_W$-M-level-inv S* ⟹ *init-clss S = init-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$_W$-learned-clause* (*S* :: *'st*) ⟷
(*init-clss S* ⊨*psm learned-clss S*
∧ (∀ *T*. *conflicting S = Some T* ⟶ *init-clss S* ⊨*pm T*)
∧ *set* (*get-all-mark-of-propagated* (*trail S*)) ⊆ *set-mset* (*clauses S*))

of Weidenbach's book for the inital state and some additional structural properties about the trail.

lemma *cdcl$_W$-learned-clause-S0-cdcl$_W$*[*simp*]:
cdcl$_W$-learned-clause (*init-state N*)
⟨*proof*⟩

Item 4 page 81 of Weidenbach's book

lemma *cdcl$_W$-learned-clss*:
assumes
cdcl$_W$ S S′ **and**
learned: *cdcl$_W$-learned-clause S* **and**
lev-inv: *cdcl$_W$-M-level-inv S*
shows *cdcl$_W$-learned-clause S′*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-learned-clss*:
assumes
cdcl$_W$** S S′ **and**
cdcl$_W$-M-level-inv S
cdcl$_W$-learned-clause S
shows *cdcl$_W$-learned-clause S′*
⟨*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* $S' \longleftrightarrow$ (
  $(\forall\, T.\ \textit{conflicting}\ S' = \textit{Some}\ T \longrightarrow \textit{atms-of}\ T \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S'))$
 $\wedge$ $(\forall\, L\ \textit{mark}.\ \textit{Propagated}\ L\ \textit{mark} \in \textit{set}\ (\textit{trail}\ S')$
    $\longrightarrow \textit{atms-of mark} \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S'))$
 $\wedge$ $\textit{atms-of-mm}\ (\textit{learned-clss}\ S') \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S')$
 $\wedge$ $\textit{atm-of}\ `\ (\textit{lits-of-l}\ (\textit{trail}\ S')) \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S'))$

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 $(\forall\, L\ \textit{mark}.\ \textit{Propagated}\ L\ \textit{mark} \in \textit{set}\ (\textit{trail}\ S)$
   $\longrightarrow \textit{atms-of mark} \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S))$
 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*:
 $C + \{\#L\#\} \in\# A \Longrightarrow x \in \textit{atms-of}\ C \Longrightarrow x \in \textit{atms-of-mm}\ A$
 ⟨*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*:
 $x \in \textit{atms-of}\ (\textit{remove1-mset}\ L\ C) \Longrightarrow x \in \textit{atms-of}\ C$
 ⟨*proof*⟩

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*:
 $\textit{atms-of-mm}\ (\textit{learned-clss}\ S) \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S) \Longrightarrow$
  $x \in \textit{atms-of-mm}\ (\textit{learned-clss}\ T) \Longrightarrow$
  $\textit{learned-clss}\ T \subseteq\#\ \textit{learned-clss}\ S \Longrightarrow$
  $x \in \textit{atms-of-mm}\ (\textit{init-clss}\ S)$
 ⟨*proof*⟩

lemma $\textit{cdcl}_W$*-no-strange-atm-explicit*:
 assumes
  $\textit{cdcl}_W\ S\ S'$ **and**
  lev: $\textit{cdcl}_W$*-M-level-inv S* **and**
  conf: $\forall\, T.\ \textit{conflicting}\ S = \textit{Some}\ T \longrightarrow \textit{atms-of}\ T \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S)$ **and**
  decided: $\forall\, L\ \textit{mark}.\ \textit{Propagated}\ L\ \textit{mark} \in \textit{set}\ (\textit{trail}\ S)$
   $\longrightarrow \textit{atms-of mark} \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S)$ **and**
  learned: $\textit{atms-of-mm}\ (\textit{learned-clss}\ S) \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S)$ **and**
  trail: $\textit{atm-of}\ `\ (\textit{lits-of-l}\ (\textit{trail}\ S)) \subseteq \textit{atms-of-mm}\ (\textit{init-clss}\ S)$
 shows

$(\forall\, T.\ \mathit{conflicting}\ S' = \mathit{Some}\ T \longrightarrow \mathit{atms\text{-}of}\ T \subseteq \mathit{atms\text{-}of\text{-}mm}\ (\mathit{init\text{-}clss}\ S')) \wedge$
$(\forall\, L\ \mathit{mark}.\ \mathit{Propagated}\ L\ \mathit{mark} \in \mathit{set}\ (\mathit{trail}\ S')$
$\longrightarrow \mathit{atms\text{-}of}\ \mathit{mark} \subseteq \mathit{atms\text{-}of\text{-}mm}\ (\mathit{init\text{-}clss}\ S')) \wedge$
$\mathit{atms\text{-}of\text{-}mm}\ (\mathit{learned\text{-}clss}\ S') \subseteq \mathit{atms\text{-}of\text{-}mm}\ (\mathit{init\text{-}clss}\ S') \wedge$
$\mathit{atm\text{-}of}\ `\ (\mathit{lits\text{-}of\text{-}l}\ (\mathit{trail}\ S')) \subseteq \mathit{atms\text{-}of\text{-}mm}\ (\mathit{init\text{-}clss}\ S')$
$(\textbf{is}\ ?C\ S' \wedge\ ?M\ S' \wedge\ ?U\ S' \wedge\ ?V\ S')$
⟨*proof*⟩

lemma $\mathit{cdcl_W\text{-}no\text{-}strange\text{-}atm\text{-}inv}$:
assumes $\mathit{cdcl_W}\ S\ S'$ **and** $\mathit{no\text{-}strange\text{-}atm}\ S$ **and** $\mathit{cdcl_W\text{-}M\text{-}level\text{-}inv}\ S$
shows $\mathit{no\text{-}strange\text{-}atm}\ S'$
⟨*proof*⟩

lemma $\mathit{rtranclp\text{-}cdcl_W\text{-}no\text{-}strange\text{-}atm\text{-}inv}$:
assumes $\mathit{cdcl_W}^{**}\ S\ S'$ **and** $\mathit{no\text{-}strange\text{-}atm}\ S$ **and** $\mathit{cdcl_W\text{-}M\text{-}level\text{-}inv}\ S$
shows $\mathit{no\text{-}strange\text{-}atm}\ S'$
⟨*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 $\mathit{distinct\text{-}cdcl_W\text{-}state}\ (S ::'st)$
$\longleftrightarrow ((\forall\, T.\ \mathit{conflicting}\ S = \mathit{Some}\ T \longrightarrow \mathit{distinct\text{-}mset}\ T)$
$\wedge\ \mathit{distinct\text{-}mset\text{-}mset}\ (\mathit{learned\text{-}clss}\ S)$
$\wedge\ \mathit{distinct\text{-}mset\text{-}mset}\ (\mathit{init\text{-}clss}\ S)$
$\wedge\ (\forall\, L\ \mathit{mark}.\ (\mathit{Propagated}\ L\ \mathit{mark} \in \mathit{set}\ (\mathit{trail}\ S) \longrightarrow \mathit{distinct\text{-}mset}\ \mathit{mark})))$

lemma $\mathit{distinct\text{-}cdcl_W\text{-}state\text{-}decomp}$:
assumes $\mathit{distinct\text{-}cdcl_W\text{-}state}\ (S ::'st)$
shows
$\forall\, T.\ \mathit{conflicting}\ S = \mathit{Some}\ T \longrightarrow \mathit{distinct\text{-}mset}\ T$ **and**
$\mathit{distinct\text{-}mset\text{-}mset}\ (\mathit{learned\text{-}clss}\ S)$ **and**
$\mathit{distinct\text{-}mset\text{-}mset}\ (\mathit{init\text{-}clss}\ S)$ **and**
$\forall\, L\ \mathit{mark}.\ (\mathit{Propagated}\ L\ \mathit{mark} \in \mathit{set}\ (\mathit{trail}\ S) \longrightarrow \mathit{distinct\text{-}mset}\ \mathit{mark})$
⟨*proof*⟩

lemma $\mathit{distinct\text{-}cdcl_W\text{-}state\text{-}decomp\text{-}2}$:
assumes $\mathit{distinct\text{-}cdcl_W\text{-}state}\ (S ::'st)$ **and** $\mathit{conflicting}\ S = \mathit{Some}\ T$
shows $\mathit{distinct\text{-}mset}\ T$
⟨*proof*⟩

lemma $\mathit{distinct\text{-}cdcl_W\text{-}state\text{-}S0\text{-}cdcl_W}\ [\mathit{simp}]$:
$\mathit{distinct\text{-}mset\text{-}mset}\ N \Longrightarrow \mathit{distinct\text{-}cdcl_W\text{-}state}\ (\mathit{init\text{-}state}\ N)$
⟨*proof*⟩

lemma $\mathit{distinct\text{-}cdcl_W\text{-}state\text{-}inv}$:
assumes
$\mathit{cdcl_W}\ S\ S'$ **and**
$\mathit{lev\text{-}inv}$: $\mathit{cdcl_W\text{-}M\text{-}level\text{-}inv}\ S$ **and**
$\mathit{distinct\text{-}cdcl_W\text{-}state}\ S$
shows $\mathit{distinct\text{-}cdcl_W\text{-}state}\ S'$
⟨*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#}) $\wedge$ $L \in\#$ *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#}) $\wedge$ $L \in\#$ *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**
∀ *T*. *conflicting* (*init-state N*) = *Some T* ⟶ (*trail* (*init-state N*))⊨*as CNot T* **and**
no-strange-atm (*init-state N*) **and**
consistent-interp (*lits-of-l* (*trail* (*init-state N*))) **and**
∀ *L mark a b*. *a* @ *Propagated L mark* # *b* = *trail* (*init-state N*) ⟶
(*b* ⊨*as CNot* (*mark* − {#*L*#}) ∧ *L* ∈# *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: ∀ *x* ∈ *set M*. ¬ *is-decided x* **and**
DN: *D* ∈# *clauses S* **and**
D: *M* ⊨*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 ∀ *m* ∈ *set M*. ¬*is-decided m*
  shows *set-mset N* ⊨*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: ∀ *s* ∈# *learned-clss S*. ¬*tautology s*
  shows ∀ *s* ∈# *learned-clss S′*. ¬*tautology s*
  ⟨*proof*⟩

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

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

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*) ⊆ *atms-of-mm N*

shows $\exists S.\ rtranclp\ cdcl_W\ (init\text{-}state\ N)\ S$
$\wedge\ state\ S = (map\ (\lambda L.\ Decided\ L)\ M,\ N,\ \{\#\},\ length\ M,\ None)$
⟨*proof*⟩

theorem 2.9.11 page 84 of Weidenbach's book

lemma $cdcl_W\text{-}strong\text{-}completeness$:
assumes
MN: $set\ M \models sm\ N$ **and**
$cons$: $consistent\text{-}interp\ (set\ M)$ **and**
$dist$: $distinct\ M$ **and**
atm: $atm\text{-}of\ `(set\ M) \subseteq atms\text{-}of\text{-}mm\ N$
obtains S **where**
$state\ S = (map\ (\lambda L.\ Decided\ L)\ M,\ N,\ \{\#\},\ length\ M,\ None)$ **and**
$rtranclp\ cdcl_W\ (init\text{-}state\ N)\ S$ **and**
$final\text{-}cdcl_W\text{-}state\ S$
⟨*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 $tranclp\text{-}conflict$:
$tranclp\ conflict\ S\ S' \Longrightarrow conflict\ S\ S'$
⟨*proof*⟩

lemma $tranclp\text{-}conflict\text{-}iff[iff]$:
$full1\ conflict\ S\ S' \longleftrightarrow conflict\ S\ S'$
⟨*proof*⟩

inductive $cdcl_W\text{-}cp :: 'st \Rightarrow 'st \Rightarrow bool$ **where**
$conflict'[intro]$: $conflict\ S\ S' \Longrightarrow cdcl_W\text{-}cp\ S\ S'$ |
$propagate'$: $propagate\ S\ S' \Longrightarrow cdcl_W\text{-}cp\ S\ S'$

lemma $rtranclp\text{-}cdcl_W\text{-}cp\text{-}rtranclp\text{-}cdcl_W$:
$cdcl_W\text{-}cp^{**}\ S\ T \Longrightarrow cdcl_W{}^{**}\ S\ T$
⟨*proof*⟩

lemma $cdcl_W\text{-}cp\text{-}state\text{-}eq\text{-}compatible$:
assumes
$cdcl_W\text{-}cp\ S\ T$ **and**
$S \sim S'$ **and**
$T \sim T'$
shows $cdcl_W\text{-}cp\ S'\ T'$
⟨*proof*⟩

lemma $tranclp\text{-}cdcl_W\text{-}cp\text{-}state\text{-}eq\text{-}compatible$:
assumes
$cdcl_W\text{-}cp^{++}\ S\ T$ **and**
$S \sim S'$ **and**
$T \sim T'$
shows $cdcl_W\text{-}cp^{++}\ S'\ T'$
⟨*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 $\Longrightarrow$ ¬*conflict T U*
⟨*proof*⟩

lemma *no-propagate-after-conflict*:
conflict S T $\Longrightarrow$ ¬*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*)
∨ (∃ *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* $\Longrightarrow$ ¬*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* ⇒ *'st* ⇒ *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** (∀ *l* ∈ *set M*. ¬*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** ∀ *l* ∈ *set M*. ¬*is-decided l*
⟨*proof*⟩

lemma *cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp S S′*
shows ∃ *M*. *trail S′* = *M* @ *trail S* ∧ (∀ *l* ∈ *set M*. ¬*is-decided l*)
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp*** *S S′*
shows ∃ *M*. *trail S′* = *M* @ *trail S* ∧ (∀ *l* ∈ *set M*. ¬*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*) ≤ *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 (λ*S*. *card* (*atms-of-mm* (*init-clss S*)) − *length* (*trail S*)
+ (*if conflicting S* = *None then 1 else 0*)) *S*
> (λ*S*. *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* ∧ *no-strange-atm a*) ∧ *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 (λ*a b*. (*cdcl$_W$-M-level-inv a* ∧ *no-strange-atm a*) ∧ *cdcl$_W$-cp a b*)** *S T*
⟷ *cdcl$_W$-cp*** *S T*
(**is** *?I S T* ⟷ *?C S T*)
⟨*proof*⟩

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 ∃ *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* ⇒ *bool* **where**
no-clause-is-false ≡
λ*S*. (*conflicting S* = *None* ⟶ (∀ *D* ∈# *clauses S*. ¬*trail S* ⊨*as CNot D*))

abbreviation *conflict-is-false-with-level* :: *'st* ⇒ *bool* **where**
conflict-is-false-with-level S ≡ ∀ *D*. *conflicting S* = *Some D* ⟶ *D* ≠ {#}
⟶ (∃ *L* ∈# *D*. *get-level* (*trail S*) *L* = *backtrack-lvl S*)

lemma *not-conflict-not-any-negated-init-clss*:
assumes ∀ *S′*. ¬*conflict S S′*
shows *no-clause-is-false S*
⟨*proof*⟩

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* ∨ (∃ *T*. *propagate** S T* ∧ *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′ ≡
∀ *D M1 M2 L*. *M1* @ *Propagated L D* # *M2* = *trail S′* ⟶ *D* − {#*L*#} ≠ {#}
⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S′*) *L* = *count-decided M1*)

definition *no-more-propagation-to-do* :: *'st* ⇒ *bool* **where**
no-more-propagation-to-do S ≡
∀ *D M M′ L*. *D* + {#*L*#} ∈# *clauses S* ⟶ *trail S* = *M′* @ *M* ⟶ *M* ⊨*as CNot D*
⟶ *undefined-lit M L* ⟶ *count-decided M* < *backtrack-lvl S*
⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S*) *L* = *count-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 ∃ *S′*. *cdcl$_W$-stgy S S′*
⟨*proof*⟩

lemma *backtrack-no-decomp*:
  assumes
    S: *conflicting S = Some E* **and**
    LE: *L* ∈# *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 ∃ *S′*. *cdcl$_W$-o S S′*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-final-state-conclusive*:
  assumes
    termi: ∀ *S′*. ¬*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′ ⟹ cdcl$_W$$^{++}$ S S′
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-stgy-tranclp-cdcl$_W$*:
  cdcl$_W$-stgy^{++} S S′ ⟹ cdcl$_W$$^{++}$ S S′
  ⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$*:
  cdcl$_W$-stgy** S S′ ⟹ cdcl$_W$** S S′
  ⟨*proof*⟩

lemma *not-empty-get-maximum-level-exists-lit*:
  assumes *n*: *D ≠ {#}*
  and *max*: *get-maximum-level M D = n*
  shows *∃ L ∈# 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 ∨ (∃ S′. propagate** S S′ ∧ conflict S′ T)*
  ⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-propagate-confl*:
  assumes *cdcl$_W$-cp** S T*
  shows *propagate** S T ∨ (∃ S′. propagate** S S′ ∧ 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#} ∈# local.clauses S **and**
    M′ ⊨as CNot C **and**
    undefined-lit (trail S) L
⟨*proof*⟩

lemma *cdcl$_W$-cp-propagate-completeness*:
  assumes *MN*: *set M ⊨s set-mset N* **and**
  cons: *consistent-interp (set M)* **and**
  tot: *total-over-m (set M) (set-mset N)* **and**
  lits-of-l (trail S) ⊆ set M **and**

init-clss S = N **and**
*propagate** S S'* **and**
learned-clss S = {#}
shows *length (trail S) ≤ length (trail S') ∧ lits-of-l (trail S') ⊆ set M*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *completeness-is-a-full1-propagation*:
fixes *S :: 'st* **and** *M :: 'v literal list*
assumes *MN*: *set M ⊨s set-mset N*
and *cons*: *consistent-interp (set M)*
and *tot*: *total-over-m (set M) (set-mset N)*
and *alien*: *no-strange-atm S*
and *learned*: *learned-clss S = {#}*
and *clsS*[*simp*]: *init-clss S = N*
and *lits*: *lits-of-l (trail S) ⊆ set M*
shows ∃ *S'*. *propagate** S S' ∧ full cdcl$_W$-cp S S'*
⟨*proof*⟩

See also *rtranclp-cdcl$_W$-cp-dropWhile-trail*

lemma *rtranclp-propagate-is-trail-append*:
*propagate** S T ⟹ ∃ c. trail T = c @ trail S*
⟨*proof*⟩

lemma *rtranclp-propagate-is-update-trail*:
*propagate** S T ⟹ cdcl$_W$-M-level-inv S ⟹*
init-clss S = init-clss T ∧ learned-clss S = learned-clss T ∧ backtrack-lvl S = backtrack-lvl T
∧ conflicting S = conflicting T
⟨*proof*⟩

lemma *cdcl$_W$-stgy-strong-completeness-n*:
assumes
MN: *set M ⊨s set-mset N* **and**
cons: *consistent-interp (set M)* **and**
tot: *total-over-m (set M) (set-mset N)* **and**
atm-incl: *atm-of ' (set M) ⊆ atms-of-mm N* **and**
distM: *distinct M* **and**
length: *n ≤ length M*
shows
∃ *M' k S*. *length M' ≥ n ∧*
lits-of-l M' ⊆ set M ∧
no-dup M' ∧
state S = (M', N, {#}, k, None) ∧
*cdcl$_W$-stgy** (init-state N) S*
⟨*proof*⟩

theorem 2.9.11 page 84 of Weidenbach's book (with strategy)

lemma *cdcl$_W$-stgy-strong-completeness*:
assumes
MN: *set M ⊨s set-mset N* **and**

cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset N*) **and**
atm-incl: *atm-of* ‘ (*set M*) ⊆ *atms-of-mm N* **and**
distM: *distinct M*
shows
∃ *M′ k S*.
lits-of-l M′ = *set M* ∧
state S = (*M′*, *N*, {#}, *k*, *None*) ∧
$cdcl_W$*-stgy*** (*init-state N*) *S* ∧
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*) ≡
(∀ *M K M′ D*. *M′* @ *Decided K* # *M* = *trail S* ⟶ *D* ∈# *clauses S*
⟶ ¬*M* ⊨*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 ¬(∀ *D*∈#*init-clss S′* + *learned-clss S′*. ¬ *trail S′* ⊨*as CNot D*)
and *conflicting S′* = *None*
shows ∃ *S″*. *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′*
∨ (*conflicting S′* = *None*
⟶ (∀ *D* ∈# *clauses S′*. *trail S′* ⊨*as CNot D*
⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S′*) *L* = *backtrack-lvl S′*)))
⟨*proof*⟩

lemma *full1-cdcl$_W$-cp-exists-conflict-decompose*:
assumes
confl: ∃ *D*∈#*clauses S*. *trail S* ⊨*as CNot D* **and**
full: *full cdcl$_W$-cp S U* **and**
no-confl: *conflicting S* = *None* **and**
lev: *cdcl$_W$-M-level-inv S*
shows ∃ *T*. *propagate*** *S T* ∧ *conflict T U*

⟨*proof*⟩

lemma *full1-cdcl$_W$-cp-exists-conflict-full1-decompose*:
  assumes
    confl: ∃ *D*∈#*clauses S*. *trail S* ⊨*as CNot D* **and**
    full: *full cdcl$_W$-cp S U* **and**
    no-confl: *conflicting S* = *None***and**
    lev: *cdcl$_W$-M-level-inv S*
  shows ∃ *T D*. *propagate*** *S T* ∧ *conflict T U*
    ∧ *trail T* ⊨*as CNot D* ∧ *conflicting U* = *Some D* ∧ *D* ∈# *clauses S*
⟨*proof*⟩

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′* ∧ *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 {#}* ∧ *unsatisfiable* (*set-mset* (*init-clss S′*)))
∨ (*conflicting S′ = None* ∧ *trail S′* ⊨*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 ≠ None
shows *False*
⟨*proof*⟩

lemma *cdcl$_W$-o-fst-empty-conflicting-false*:
assumes *cdcl$_W$-o S S′*
and *trail S = []*
and *conflicting S ≠ None*
shows *False*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-fst-empty-conflicting-false*:
assumes *cdcl$_W$-stgy S S′*
and *trail S = []*
and *conflicting S ≠ None*
shows *False*
⟨*proof*⟩
thm *cdcl$_W$-cp.induct*[*split-format*(*complete*)]

lemma *cdcl$_W$-cp-conflicting-is-false*:
cdcl$_W$-cp S S′ ⟹ *conflicting S = Some {#}* ⟹ *False*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-conflicting-is-false*:
cdcl$_W$-cp^{++} S S′ ⟹ *conflicting S = Some {#}* ⟹ *False*
⟨*proof*⟩

lemma *cdcl$_W$-o-conflicting-is-false*:
cdcl$_W$-o S S′ ⟹ *conflicting S = Some {#}* ⟹ *False*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-conflicting-is-false*:
cdcl$_W$-stgy S S′ ⟹ *conflicting S = Some {#}* ⟹ *False*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-stgy-conflicting-is-false*:
cdcl$_W$-stgy** S S′ ⟹ *conflicting S = Some {#}* ⟹ *S′ = S*
⟨*proof*⟩

lemma *full-cdcl$_W$-init-clss-with-false-normal-form*:

assumes
∀ *m*∈ *set M*. ¬*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 ∃ *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*: {#} ∈# *N*
shows *conflicting S′* = *Some* {#} ∧ *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* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S′*)))
∨ (*conflicting S′* = *None* ∧ *trail S′* ⊨*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* {#} ∧ *unsatisfiable* (*set-mset N*))
∨ (*conflicting S′* = *None* ∧ *trail S′* ⊨*asm N* ∧ *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 $S \wedge$
$(\forall\, s \in\# \textit{learned-clss}\ S.\ \neg \textit{tautology}\ s) \wedge$
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*: $D \in\# \textit{learned-clss}\ T$ **and**
new: $D \notin\# \textit{learned-clss}\ S$ **and**
$cdcl_W$: $cdcl_W$-o $S\ T$ **and**
lev: $cdcl_W$-M-level-inv S
shows *backtrack* $S\ T \wedge$ *conflicting* $S = \textit{Some}\ D$
⟨*proof*⟩

lemma $cdcl_W$-cp-new-clause-learned-has-backtrack-step:
assumes *learned*: $D \in\# \textit{learned-clss}\ T$ **and**
new: $D \notin\# \textit{learned-clss}\ S$ **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* $S = \textit{Some}\ D$
⟨*proof*⟩

lemma rtranclp-$cdcl_W$-cp-new-clause-learned-has-backtrack-step:
assumes *learned*: $D \in\# \textit{learned-clss}\ T$ **and**
new: $D \notin\# \textit{learned-clss}\ S$ **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* $S' = \textit{Some}\ D \wedge$
$cdcl_W$-stgy$^{**}\ S''\ T$
⟨*proof*⟩

lemma *propagate-no-more-Decided-lit*:
assumes *propagate* $S\ S'$

shows *Decided K ∈ set (trail S) ⟷ Decided K ∈ set (trail S′)*
⟨*proof*⟩

lemma *conflict-no-more-Decided-lit*:
assumes *conflict S S′*
shows *Decided K ∈ set (trail S) ⟷ Decided K ∈ set (trail S′)*
⟨*proof*⟩

lemma *cdcl$_W$-cp-no-more-Decided-lit*:
assumes *cdcl$_W$-cp S S′*
shows *Decided K ∈ set (trail S) ⟷ Decided K ∈ set (trail S′)*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-no-more-Decided-lit*:
assumes *cdcl$_W$-cp** S S′*
shows *Decided K ∈ set (trail S) ⟷ Decided K ∈ set (trail S′)*
⟨*proof*⟩

lemma *cdcl$_W$-o-no-more-Decided-lit*:
assumes *cdcl$_W$-o S S′* **and** *lev*: *cdcl$_W$-M-level-inv S* **and** *¬decide S S′*
shows *Decided K ∈ set (trail S′) ⟶ Decided K ∈ set (trail S)*
⟨*proof*⟩

lemma *cdcl$_W$-new-decided-at-beginning-is-decide*:
assumes *cdcl$_W$-stgy S S′* **and**
lev: *cdcl$_W$-M-level-inv S* **and**
trail S′ = M′ @ Decided L # M **and**
trail S = M
shows *∃ T. decide S T ∧ no-step cdcl$_W$-cp S*
⟨*proof*⟩

lemma *cdcl$_W$-o-is-decide*:
assumes *cdcl$_W$-o S T* **and** *lev*: *cdcl$_W$-M-level-inv S*
*trail T = drop (length M$_0$) M′ @ Decided L # H @ M***and**
¬ (∃ M′. trail S = M′ @ Decided L # H @ M)
shows *decide S T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-new-decided-at-beginning-is-decide*:
assumes *cdcl$_W$-stgy** R U* **and**
trail U = M′ @ Decided L # H @ M **and**
trail R = M **and**
cdcl$_W$-M-level-inv R
shows
∃ S T T′. cdcl$_W$-stgy** R S ∧ decide S T ∧ cdcl$_W$-stgy** T U ∧ cdcl$_W$-stgy** S U ∧
no-step cdcl$_W$-cp S ∧ trail T = Decided L # H @ M ∧ trail S = H @ M ∧ cdcl$_W$-stgy S T′ ∧
cdcl$_W$-stgy** T′ U
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-new-decided-at-beginning-is-decide′*:
assumes *cdcl$_W$-stgy** R U* **and**
trail U = M′ @ Decided L # H @ M **and**
trail R = M **and**
cdcl$_W$-M-level-inv R
shows *∃ y y′. cdcl$_W$-stgy** R y ∧ cdcl$_W$-stgy y y′ ∧ ¬ (∃ c. trail y = c @ Decided L # H @ M)*
∧ (λa b. cdcl$_W$-stgy a b ∧ (∃ 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: $\forall m \in set\ M.\ \neg is\text{-}decided\ m$
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**
$(\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \wedge (\exists c.\ trail\ a = c$ @ *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 $\exists S'.\ decide\ S\ S' \wedge full\ cdcl_W\text{-}cp\ S'\ T \wedge no\text{-}step\ cdcl_W\text{-}cp\ S$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-with-trail-end-has-trail-end*:
assumes $(\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \wedge (\exists c.\ trail\ a = c$ @ *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 $cdcl_W\text{-}stgy\text{-}new\text{-}learned\text{-}clause$:
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 $cdcl_W\text{-}stgy\text{-}no\text{-}relearned\text{-}clause$:
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**
$already\text{-}learned$: $E \in\# clauses\ S$ **and**
R: $trail\ R = []$
shows $False$
⟨*proof*⟩

lemma $rtranclp\text{-}cdcl_W\text{-}stgy\text{-}distinct\text{-}mset\text{-}clauses$:
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 $cdcl_W\text{-}stgy\text{-}distinct\text{-}mset\text{-}clauses$:
assumes
st: $cdcl_W\text{-}stgy^{**}\ (init\text{-}state\ N)\ S$ **and**
$no\text{-}duplicate\text{-}clause$: $distinct\text{-}mset\ N$ **and**
$no\text{-}duplicate\text{-}in\text{-}clause$: $distinct\text{-}mset\text{-}mset\ N$
shows $distinct\text{-}mset\ (clauses\ S)$
⟨*proof*⟩

Decrease of a Measure

fun $cdcl_W\text{-}measure$ **where**
$cdcl_W\text{-}measure\ S =$
$[(3::nat)\ \widehat{}\ (card\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S))) - card\ (set\text{-}mset\ (learned\text{-}clss\ S)),$
$if\ conflicting\ S = None\ then\ 1\ else\ 0,$
$if\ conflicting\ S = None\ then\ card\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S)) - length\ (trail\ S)$

else length (*trail S*)
]

lemma *length-model-le-vars-all-inv*:
assumes $cdcl_W$*-all-struct-inv S*
shows *length* (*trail S*) ≤ *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**
∀ *s* ∈# *learned-clss S*. ¬*tautology s*
shows *card*(*set-mset* (*learned-clss S*)) ≤ *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:
¬(*learned-clss S* ⊆# *learned-clss S'* ∧ [] = *trail S'* ∧ *conflicting S'* = *None*)
and
no-forget: *learned-clss S* ⊆# *learned-clss S'* **and**
no-relearn: ⋀*S'*. *backtrack S S'* ⟹ ∀ *T*. *conflicting S* = *Some T* ⟶ *T* ∉# *learned-clss S*
and
alien: *no-strange-atm S* **and**
M-level: $cdcl_W$*-M-level-inv S* **and**
no-taut: ∀ *s* ∈# *learned-clss S*. ¬*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*) ∈ *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*) ∈ *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*) ∈ *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*) ∈ *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 $\{(S::'st, init\text{-}state\ N) |$
$S\ N.$ *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. *conflict-driven-clause-learning$_W$.conflict* to find the conflict
2. the conflict is analysed by repetitive application of *conflict-driven-clause-learning$_W$.resolve* and *conflict-driven-clause-learning$_W$.skip*,
3. finally *conflict-driven-clause-learning$_W$.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 *conflict-driven-clause-learning$_W$*
begin
declare *cdcl$_W$.intros*[*intro*] *cdcl$_W$-bj.intros*[*intro*] *cdcl$_W$-o.intros*[*intro*]

lemma *backtrack-no-cdcl$_W$-bj*:
  assumes *cdcl*: *cdcl$_W$-bj T U* **and** *inv*: *cdcl$_W$-M-level-inv V*
  shows ¬*backtrack V T*
  ⟨*proof*⟩

skip-or-resolve corresponds to the *analyze* function in the code of MiniSAT.

inductive *skip-or-resolve* :: *'st* ⇒ *'st* ⇒ *bool* **where**
s-or-r-skip[*intro*]: *skip S T* ⟹ *skip-or-resolve S T* |
s-or-r-resolve[*intro*]: *resolve S T* ⟹ *skip-or-resolve S T*

lemma *rtranclp-cdcl$_W$-bj-skip-or-resolve-backtrack*:
  assumes *cdcl$_W$-bj*** *S U* **and** *inv*: *cdcl$_W$-M-level-inv S*
  shows *skip-or-resolve*** *S U* ∨ (∃ *T*. *skip-or-resolve*** *S T* ∧ *backtrack T U*)
  ⟨*proof*⟩

lemma *rtranclp-skip-or-resolve-rtranclp-cdcl$_W$*:
  *skip-or-resolve*** *S T* ⟹ *cdcl$_W$*** *S T*
  ⟨*proof*⟩

definition *backjump-l-cond* :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **where**
backjump-l-cond ≡ λ*C C' L' S T*. *True*

definition *inv$_{NOT}$* :: *'st* ⇒ *bool* **where**
inv$_{NOT}$ ≡ λ*S*. *no-dup* (*trail S*)

declare *inv$_{NOT}$-def*[*simp*]
end

context *conflict-driven-clause-learning*$_W$
begin

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*. (λ*S T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T*
∧ (λ*T U*. *resolve T U* ∧ *no-step backtrack T*) *T U*
∧ *skip** U V* ∧ *backtrack V W*)
∨ (∃ *T U*. (λ*S T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T*

$\wedge$ ($\lambda T\ U$. *resolve* $T\ U \wedge$ *no-step backtrack* T) $T\ U \wedge$ *skip*** $U\ W$)
$\vee$ ($\exists\, T$. *skip*** $S\ T \wedge$ *backtrack* $T\ W$)
$\vee$ *skip*** $S\ W$ (**is** *?RB S W* $\vee$ *?R S W* $\vee$ *?SB S W* $\vee$ *?S S W*)
⟨*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′ ⟹ cdcl$_W$-merge S S′*|
fw-forget: *forget S S′ ⟹ cdcl$_W$-merge S S′*

lemma *cdcl$_W$-merge-cdcl$_W$-merge-restart*:
cdcl$_W$-merge S T ⟹ cdcl$_W$-merge-restart S T
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-tranclp-cdcl$_W$-merge-restart*:
cdcl$_W$-merge** S T ⟹ cdcl$_W$-merge-restart** S T
⟨*proof*⟩

lemma *cdcl$_W$-merge-rtranclp-cdcl$_W$*:
cdcl$_W$-merge S T ⟹ cdcl$_W{}^{**}$ S T
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-rtranclp-cdcl$_W$*:
cdcl$_W$-merge** S T ⟹ 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 (λ*S T*. *cdcl$_W$-all-struct-inv S ∧ cdcl$_W$-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 ∧ conflicting V = None*)
∨ (∃ *T U*. *cdcl$_W$-merge-restart** S T ∧ conflicting V ≠ None ∧ conflict T U ∧ cdcl$_W$-bj** U V*)
⟨*proof*⟩

lemma *no-step-cdcl$_W$-no-step-cdcl$_W$-merge-restart*: *no-step cdcl$_W$ S ⟹ 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*

⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟷ *full cdcl$_W$-merge-restart S V*
⟨*proof*⟩

lemma *init-state-true-full-cdcl$_W$-iff-full-cdcl$_W$-merge*:
shows *full cdcl$_W$* (*init-state N*) *V* ⟷ *full cdcl$_W$-merge-restart* (*init-state N*) *V*
⟨*proof*⟩

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′* ⟹ *cdcl$_W$-s′ S S′* |
decide′: *decide S S′* ⟹ *no-step cdcl$_W$-cp S* ⟹ *full cdcl$_W$-cp S′ S″* ⟹ *cdcl$_W$-s′ S S″* |
bj′: *full1 cdcl$_W$-bj S S′* ⟹ *no-step cdcl$_W$-cp S* ⟹ *full cdcl$_W$-cp S′ S″* ⟹ *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′* ⟹ *full cdcl$_W$-cp S′ S″* ⟹ *cdcl$_W$-stgy*** *S S″*
⟨*proof*⟩

lemma *cdcl$_W$-s′-is-rtranclp-cdcl$_W$-stgy*:
cdcl$_W$-s′ S T ⟹ *cdcl$_W$-stgy*** *S T*
⟨*proof*⟩

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″*)
⟨*proof*⟩

lemma *cdcl$_W$-cp-cdcl$_W$-bj-bissimulation′*:
assumes
full cdcl$_W$-cp T U **and**
cdcl$_W$-bj** *T T′* **and**
cdcl$_W$-all-struct-inv T **and**

no-step cdcl$_W$-bj T'
shows *full cdcl$_W$-cp T' U*
∨ (∃ *U'*. *full1 cdcl$_W$-bj U U'* ∧ (∀ *U''*. *full cdcl$_W$-cp U' U''* ⟶ *full cdcl$_W$-cp T' U''*
∧ *cdcl$_W$-s'** U U''*))
⟨*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*
∨ (∃ *U'*. *full1 cdcl$_W$-bj U U'* ∧ (∀ *U''*. *full cdcl$_W$-cp U' U''* ⟶ *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*
∨ (∃ *U' U''*. *cdcl$_W$-s' S U''* ∧ *full1 cdcl$_W$-bj U U'* ∧ *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* ∨ (∃ *T*. *cdcl$_W$-s'** S T* ∧ *cdcl$_W$-bj^{++} T U* ∧ *conflicting U* ≠ *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* ⟷ *no-step cdcl$_W$-cp S* ∧ *no-step cdcl$_W$-o S* (**is** *?S' S* ⟷ *?C S* ∧ *?O S*)
⟨*proof*⟩

lemma *cdcl$_W$-s'-tranclp-cdcl$_W$*:
cdcl$_W$-s' S S' ⟹ *cdcl$_W$$^{++}$ S S'*
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-s'-tranclp-cdcl$_W$*:
cdcl$_W$-s'$^{++}$ S S' ⟹ *cdcl$_W$$^{++}$ S S'*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s'-rtranclp-cdcl$_W$*:
cdcl$_W$-s'** S S' ⟹ *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* ⟷ *full cdcl$_W$-s' S T* (**is** *?S* ⟷ *?S'*)
⟨*proof*⟩

lemma *conflict-step-cdcl$_W$-stgy-step*:
assumes
conflict S T
cdcl$_W$-all-struct-inv S
shows ∃ *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

$(cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ V)$
$\vee\ (\exists\ T.\ cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ T \wedge propagate^{++}\ T\ V)$
$\vee\ (\exists\ T\ U.\ cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ T \wedge full1\ cdcl_W\text{-}bj\ T\ U \wedge propagate^{**}\ U\ V)$
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s′-without-decide-is-rtranclp-cdcl$_W$-merge-cp*:
assumes
cdcl$_W$-s′-without-decide** S V **and**
confl: *conflicting S = None*
shows
$(cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V \wedge conflicting\ V = None)$
$\vee\ (cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V \wedge conflicting\ V \neq None \wedge no\text{-}step\ cdcl_W\text{-}cp\ V \wedge no\text{-}step\ cdcl_W\text{-}bj\ V)$
$\vee\ (\exists\ T.\ cdcl_W\text{-}merge\text{-}cp^{**}\ S\ T \wedge conflict\ T\ V)$
⟨*proof*⟩

lemma *no-step-cdcl$_W$-s′-no-ste-cdcl$_W$-merge-cp*:
assumes
cdcl$_W$-all-struct-inv S
conflicting S = None
no-step cdcl$_W$-s′ S
shows *no-step cdcl$_W$-merge-cp S*
⟨*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$_W$-M-level-inv S* **and**
n-s: *no-step cdcl$_W$-merge-cp S*
shows *no-step cdcl$_W$-s′-without-decide S*
⟨*proof*⟩

lemma *conflicting-true-no-step-s′-without-decide-no-step-cdcl$_W$-merge-cp*:
assumes
inv: *cdcl$_W$-all-struct-inv S* **and**
n-s: *no-step cdcl$_W$-s′-without-decide S*
shows *no-step cdcl$_W$-merge-cp S*
⟨*proof*⟩

lemma *no-step-cdcl$_W$-merge-cp-no-step-cdcl$_W$-cp*:
no-step cdcl$_W$-merge-cp S $\Longrightarrow$ *cdcl$_W$-M-level-inv S* $\Longrightarrow$ *no-step cdcl$_W$-cp S*
⟨*proof*⟩

lemma *conflicting-not-true-rtranclp-cdcl$_W$-merge-cp-no-step-cdcl$_W$-bj*:
assumes
conflicting S = None **and**
cdcl$_W$-merge-cp** S T
shows *no-step cdcl$_W$-bj T*
⟨*proof*⟩

lemma *conflicting-true-full-cdcl$_W$-merge-cp-iff-full-cdcl$_W$-s′-without-decode*:
assumes
confl: *conflicting S = None* **and**
inv: *cdcl$_W$-all-struct-inv S*
shows
full cdcl$_W$-merge-cp S V $\longleftrightarrow$ *full cdcl$_W$-s′-without-decide S V* (**is** *?fw* $\longleftrightarrow$ *?s′*)

⟨*proof*⟩

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*

⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s'-w-rtranclp-cdcl$_W$*:
cdcl$_W$-s'-w** S T ⟹ *cdcl$_W$** S T*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

lemma *no-step-cdcl$_W$-s'-without-decide-no-step-cdcl$_W$-cp*:
cdcl$_W$-all-struct-inv S ⟹ *no-step cdcl$_W$-s'-without-decide S* ⟹ *no-step cdcl$_W$-cp S*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s'-w-no-step-cdcl$_W$-cp-or-eq*:
assumes *cdcl$_W$-s'-w** S T* **and** *cdcl$_W$-all-struct-inv S*
shows *S = T* ∨ *no-step cdcl$_W$-cp T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-stgy'-no-step-cdcl$_W$-cp-or-eq*:
assumes *cdcl$_W$-merge-stgy** S T* **and** *inv*: *cdcl$_W$-all-struct-inv S*
shows *S = T* ∨ *no-step cdcl$_W$-cp T*
⟨*proof*⟩

lemma *no-step-cdcl$_W$-s'-without-decide-no-step-cdcl$_W$-bj*:
assumes *no-step cdcl$_W$-s'-without-decide S* **and** *inv*: *cdcl$_W$-all-struct-inv S*
shows *no-step cdcl$_W$-bj S*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s'-w-no-step-cdcl$_W$-bj-or-eq*:
assumes *cdcl$_W$-s'-w** S T* **and** *cdcl$_W$-all-struct-inv S*
shows *S = T* ∨ *no-step cdcl$_W$-bj T*
⟨*proof*⟩

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: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$
shows $(cdcl_W\text{-}merge\text{-}stgy^{**}\ R\ V \wedge conflicting\ V = None)$
$\vee\ (cdcl_W\text{-}merge\text{-}stgy^{**}\ R\ V \wedge conflicting\ V \neq None \wedge no\text{-}step\ cdcl_W\text{-}bj\ V)$
$\vee\ (\exists S\ T\ U.\ cdcl_W\text{-}merge\text{-}stgy^{**}\ R\ S \wedge no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S \wedge decide\ S\ T$
$\wedge\ cdcl_W\text{-}merge\text{-}cp^{**}\ T\ U \wedge conflict\ U\ V)$
$\vee\ (\exists S\ T.\ cdcl_W\text{-}merge\text{-}stgy^{**}\ R\ S \wedge no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S \wedge decide\ S\ T$
$\wedge\ cdcl_W\text{-}merge\text{-}cp^{**}\ T\ V$
$\wedge\ conflicting\ V = None)$
$\vee\ (cdcl_W\text{-}merge\text{-}cp^{**}\ R\ V \wedge conflicting\ V = None)$
$\vee\ (\exists U.\ cdcl_W\text{-}merge\text{-}cp^{**}\ R\ U \wedge conflict\ U\ V)$
⟨*proof*⟩

lemma $decide\text{-}rtranclp\text{-}cdcl_W\text{-}s'\text{-}rtranclp\text{-}cdcl_W\text{-}s'$:
assumes
dec: $decide\ S\ T$ **and**
$cdcl_W\text{-}s'^{**}\ T\ U$ **and**
$n\text{-}s\text{-}S$: $no\text{-}step\ cdcl_W\text{-}cp\ S$ **and**
$no\text{-}step\ cdcl_W\text{-}cp\ U$
shows $cdcl_W\text{-}s'^{**}\ S\ U$
⟨*proof*⟩

lemma $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W\text{-}s'$:
assumes
$cdcl_W\text{-}merge\text{-}stgy^{**}\ R\ V$ **and**
inv: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$
shows $cdcl_W\text{-}s'^{**}\ R\ V$
⟨*proof*⟩

lemma $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}distinct\text{-}mset\text{-}clauses$:
assumes $invR$: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$ **and**
st: $cdcl_W\text{-}merge\text{-}stgy^{**}\ R\ S$ **and**
$dist$: $distinct\text{-}mset\ (clauses\ R)$ **and**
R: $trail\ R = []$
shows $distinct\text{-}mset\ (clauses\ S)$
⟨*proof*⟩

lemma $no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}no\text{-}step\text{-}cdcl_W\text{-}merge\text{-}stgy$:
assumes
inv: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$ **and** s': $no\text{-}step\ cdcl_W\text{-}s'\ R$
shows $no\text{-}step\ cdcl_W\text{-}merge\text{-}stgy\ R$
⟨*proof*⟩
end

Termination and full Equivalence

We will discharge the assumption later using NOT's proof of termination.

locale $conflict\text{-}driven\text{-}clause\text{-}learning_W\text{-}termination$ =
$conflict\text{-}driven\text{-}clause\text{-}learning_W$ +
assumes $wf\text{-}cdcl_W\text{-}merge\text{-}inv$: $wf\ \{(T,\ S).\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \wedge cdcl_W\text{-}merge\ S\ T\}$
begin

lemma $wf\text{-}tranclp\text{-}cdcl_W\text{-}merge$: $wf\ \{(T,\ S).\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \wedge cdcl_W\text{-}merge^{++}\ S\ T\}$
⟨*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-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-driven-clause-learning*$_W \subseteq$ *cdcl*$_{NOT}$*-merge-bj-learn*
$\lambda S.$ *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* $C\ S$
backjump-l-cond
λ- -. *True*
λ- $S.$ *conflicting* $S = None$ inv_{NOT}
⟨*proof*⟩

context *conflict-driven-clause-learning*$_W$
begin

Notations are lost while proving locale inclusion:

notation *state-eq*$_{NOT}$ (**infix** $\sim_{NOT}$ *50*)

6.3.2 Additional Lemmas between NOT and W states

lemma *trail*$_W$*-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-learned-cls*:
no-dup (*trail* S) $\Longrightarrow$
trail (*reduce-trail-to*$_{NOT}$ M (*add-learned-cls* $D\ S$)) = *trail* (*reduce-trail-to*$_{NOT}$ $M\ S$)
⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-reduce-trail-convert*:
reduce-trail-to$_{NOT}$ $C\ S$ = *reduce-trail-to* (*convert-trail-from-NOT* C) S
⟨*proof*⟩

lemma *reduce-trail-to-map*[*simp*]:
reduce-trail-to (*map* $f\ M$) S = *reduce-trail-to* $M\ S$
⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-map*[*simp*]:
reduce-trail-to$_{NOT}$ (*map* $f\ M$) S = *reduce-trail-to*$_{NOT}$ $M\ S$
⟨*proof*⟩

lemma *skip-or-resolve-state-change*:
assumes *skip-or-resolve*** $S\ T$
shows
$\exists M.$ *trail* $S = M$ @ *trail* $T \wedge (\forall m \in$ *set* $M.\ \neg$*is-decided* $m)$
clauses S = *clauses* T
backtrack-lvl S = *backtrack-lvl* T
⟨*proof*⟩

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*
    ∨ (*no-step cdcl$_W$-merge T* ∧ *conflicting T* ≠ *None*)
  ⟨*proof*⟩

abbreviation *cdcl$_{NOT}$-restart* **where**
cdcl$_{NOT}$-restart ≡ *restart-ops.cdcl$_{NOT}$-raw-restart cdcl$_{NOT}$ restart*

lemma *cdcl$_W$-merge-restart-is-cdcl$_{NOT}$-merged-bj-learn-restart-no-step*:
  assumes
    inv: *cdcl$_W$-all-struct-inv S* **and**
    cdcl$_W$:*cdcl$_W$-merge-restart S T*
  shows *cdcl$_{NOT}$-restart*** *S T* ∨ (*no-step cdcl$_W$-merge T* ∧ *conflicting T* ≠ *None*)
⟨*proof*⟩

abbreviation μ_{FW} :: *'st* ⇒ *nat* **where**
μ_{FW} *S* ≡ (*if no-step cdcl$_W$-merge S then 0 else 1+*μ_{CDCL}*'-merged* (*set-mset* (*init-clss S*)) *S*)

lemma *cdcl$_W$-merge-*μ_{FW}*-decreasing*:
  assumes
    inv: *cdcl$_W$-all-struct-inv S* **and**
    fw: *cdcl$_W$-merge S T*
  shows μ_{FW} *T* < μ_{FW} *S*
⟨*proof*⟩

lemma *wf-cdcl$_W$-merge*: *wf* {(*T*, *S*). *cdcl$_W$-all-struct-inv S* ∧ *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* ⟷ *full cdcl$_W$-merge-stgy R V* (**is** *?s'* ⟷ *?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* ⟷ *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$_W$-merge-stgy*^^(*card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)))) *S T*
⟹ *card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)) > *f n*
⟹ *restart T U* ⟹ *cdcl$_W$-merge-with-restart* (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1 cdcl$_W$-merge-stgy S T* ⟹ *cdcl$_W$-merge-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma *cdcl$_W$-merge-with-restart S T* ⟹ *cdcl$_W$-merge-restart*** (*fst S*) (*fst T*)

⟨*proof*⟩

lemma *cdcl$_W$-merge-with-restart-rtranclp-cdcl$_W$*:
cdcl$_W$-merge-with-restart S T ⟹ *cdcl$_W$*** (*fst S*) (*fst T*)
⟨*proof*⟩

lemma *cdcl$_W$-merge-with-restart-increasing-number*:
cdcl$_W$-merge-with-restart S T ⟹ *snd T* = 1 + *snd S*
⟨*proof*⟩

lemma *full1 cdcl$_W$-merge-stgy S T* ⟹ *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*) ⊆ *simple-clss* (*atms-of-mm* (*init-clss S*))
⟨*proof*⟩

lemma *cdcl$_W$-merge-with-restart-init-clss*:
cdcl$_W$-merge-with-restart S T ⟹ *cdcl$_W$-M-level-inv* (*fst S*) ⟹
init-clss (*fst S*) = *init-clss* (*fst T*)
⟨*proof*⟩

lemma
wf {(*T*, *S*). *cdcl$_W$-all-struct-inv* (*fst S*) ∧ *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*$^{\frown\frown}$(*card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)))) *S T* ⟹
card (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)) > *f n* ⟹
restart T U ⟹
cdcl$_W$-with-restart (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1 cdcl$_W$-stgy S T* ⟹ *cdcl$_W$-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma *cdcl$_W$-with-restart-rtranclp-cdcl$_W$*:
cdcl$_W$-with-restart S T ⟹ *cdcl$_W$*** (*fst S*) (*fst T*)
⟨*proof*⟩

lemma *cdcl$_W$-with-restart-increasing-number*:
cdcl$_W$-with-restart S T ⟹ *snd T* = 1 + *snd S*
⟨*proof*⟩

lemma *full1 cdcl$_W$-stgy S T* ⟹ *cdcl$_W$-with-restart* (*S*, *n*) (*T*, *Suc n*)
⟨*proof*⟩

lemma *cdcl$_W$-with-restart-init-clss*:
cdcl$_W$-with-restart S T ⟹ *cdcl$_W$-M-level-inv* (*fst S*) ⟹ *init-clss* (*fst S*) = *init-clss* (*fst T*)

⟨*proof*⟩

lemma
wf {(*T*, *S*). *cdcl*$_W$*-all-struct-inv* (*fst S*) ∧ *cdcl*$_W$*-with-restart S T*}
⟨*proof*⟩

lemma *cdcl*$_W$*-with-restart-distinct-mset-clauses*:
assumes *invR*: *cdcl*$_W$*-all-struct-inv* (*fst R*) **and**
st: *cdcl*$_W$*-with-restart R S* **and**
dist: *distinct-mset* (*clauses* (*fst R*)) **and**
R: *trail* (*fst R*) = []
shows *distinct-mset* (*clauses* (*fst S*))
⟨*proof*⟩
end

locale *luby-sequence* =
fixes *ur* :: *nat*
assumes *ur* > *0*
begin

lemma *exists-luby-decomp*:
fixes *i* ::*nat*
shows ∃ *k*::*nat*. (*2* ^ (*k* − *1*) ≤ *i* ∧ *i* < *2* ^ *k* − *1*) ∨ *i* = *2* ^ *k* − *1*
⟨*proof*⟩

Luby sequences are defined by:

- $2^k - 1$, if $i = (2::'a)^k - (1::'a)$
- *luby-sequence-core* $(i - 2^{k-1} + 1)$, if $(2::'a)^{k-1} \leq i$ and $i \leq (2::'a)^k - (1::'a)$

Then the sequence is then scaled by a constant unit run (called *ur* here), strictly positive.

function *luby-sequence-core* :: *nat* ⇒ *nat* **where**
luby-sequence-core i =
(*if* ∃ *k*. *i* = *2*^*k* − *1*
then *2*^((*SOME k*. *i* = *2*^*k* − *1*) − *1*)
else luby-sequence-core (*i* − *2*^((*SOME k*. *2*^(*k*−*1*)≤ *i* ∧ *i* < *2*^*k* − *1*) − *1*) + *1*))
⟨*proof*⟩
termination
⟨*proof*⟩

function *natlog2* :: *nat* ⇒ *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*) ^ (*k*::*nat*) − *1* = *2* ^ *k'* − *1*
shows *k'* = *k*
⟨*proof*⟩

lemma *luby-sequence-core-two-power-minus-one*:
luby-sequence-core (*2^k − 1*) = *2^*(*k−1*) (**is** *?L* = *?K*)
⟨*proof*⟩

lemma *different-luby-decomposition-false*:
assumes
H: *2* ^ (*k − Suc 0*) ≤ *i* **and**
k': *i* < *2* ^ *k' − Suc 0* **and**
k-k': *k* > *k'*
shows *False*
⟨*proof*⟩

lemma *luby-sequence-core-not-two-power-minus-one*:
assumes
k-i: *2* ^ (*k − 1*) ≤ *i* **and**
i-k: *i* < *2^k − 1*
shows *luby-sequence-core i* = *luby-sequence-core* (*i − 2* ^ (*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*:
∃ *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* ⊨*as CNot C*
shows
(*trail* ((*cut-trail-wrt-clause C* (*trail T*) *T*))) ⊨*as CNot C*
⟨*proof*⟩

lemma *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*:
((∀ *L* ∈# *C*. −*L* ∉ *lits-of-l* (*trail T*)) ∧ *trail* (*cut-trail-wrt-clause C* (*trail T*) *T*) = [])
∨ (−*lit-of* (*hd* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))) ∈# *C*
∧ *length* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)) ≥ *1*)
⟨*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* ⇒ *'st* ⇒ *bool* **for** *S* **where**
add-confl:
trail S ⊨*asm init-clss S* ⟹ *distinct-mset C* ⟹ *conflicting S* = *None* ⟹
trail S ⊨*as CNot C* ⟹
full cdcl$_W$*-stgy*
(*update-conflicting* (*Some C*)
(*add-init-cls C* (*cut-trail-wrt-clause C* (*trail S*) *S*))) *T* ⟹
incremental-cdcl$_W$ *S T* |
add-no-confl:
trail S ⊨*asm init-clss S* ⟹ *distinct-mset C* ⟹ *conflicting S* = *None* ⟹
¬*trail S* ⊨*as CNot C* ⟹
full cdcl$_W$*-stgy* (*add-init-cls C S*) *T* ⟹

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 ⊨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*]:
$(\forall\, c \in set\ C.\ -c \in lits\text{-}of\text{-}l\ Ms) \longleftrightarrow Ms \models as\ CNot\ (mset\ C)$
⟨*proof*⟩

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: *'a literal list* ⇒ (*'a*, *'b*) *ann-lits* ⇒ *'a literal option*
where
is-unit-clause l M =
(*case List.filter* (λ*a*. *atm-of a* ∉ *atm-of* ' *lits-of-l M*) *l of*
a # [] ⇒ *if M* ⊨*as CNot* (*mset l* − {#*a*#}) *then Some a else None*
| - ⇒ *None*)

definition *is-unit-clause-code* :: *'a literal list* ⇒ (*'a*, *'b*) *ann-lits*
⇒ *'a literal option* **where**
is-unit-clause-code l M =
(*case List.filter* (λ*a*. *atm-of a* ∉ *atm-of* ' *lits-of-l M*) *l of*
a # [] ⇒ *if* (∀ *c* ∈*set* (*remove1 a l*). −*c* ∈ *lits-of-l M*) *then Some a else None*
| - ⇒ *None*)

lemma *is-unit-clause-is-unit-clause-code*[*code*]:
is-unit-clause l M = *is-unit-clause-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 ⟹ M ⊨as CNot (mset l − {#a#})*
  ⟨*proof*⟩

lemma *is-unit-clause-some-in*: *is-unit-clause l M = Some a ⟹ a ∈ 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 ⇒ ('a, 'b) ann-lits*
  ⇒ ('a literal × 'a literal list) option **where**
find-first-unit-clause (a # l) M =
  (*case is-unit-clause a M of*
    None ⇒ find-first-unit-clause l M
  | *Some L ⇒ Some (L, a)*) |
find-first-unit-clause [] - = None

lemma *find-first-unit-clause-some*:
  find-first-unit-clause l M = Some (a, c)
  ⟹ c ∈ set l ∧ M ⊨as CNot (mset c − {#a#}) ∧ undefined-lit M a ∧ a ∈ set c
  ⟨*proof*⟩

lemma *propagate-is-unit-clause-not-None*:
  assumes *dist*: *distinct c* **and**
  M: *M ⊨as CNot (mset c − {#a#})* **and**
  undef: *undefined-lit M a* **and**
  ac: *a ∈ set c*
  shows *is-unit-clause c M ≠ None*
⟨*proof*⟩

lemma *find-first-unit-clause-none*:
  distinct c ⟹ c ∈ set l ⟹ M ⊨as CNot (mset c − {#a#}) ⟹ undefined-lit M a ⟹ a ∈ set c
  ⟹ find-first-unit-clause l M ≠ None
  ⟨*proof*⟩

Decide

fun *find-first-unused-var* :: *'a literal list list ⇒ 'a literal set ⇒ 'a literal option* **where**
find-first-unused-var (a # l) M =
  (*case List.find (λlit. lit ∉ M ∧ −lit ∉ M) a of*
    None ⇒ find-first-unused-var l M
  | *Some a ⇒ Some a*) |
find-first-unused-var [] - = None

lemma *find-none*[*iff*]:

List.find ($\lambda lit.\ lit \notin M \wedge -lit \notin M$) $a = None \longleftrightarrow$ *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 × *'v clauses* × *'v clauses* × *nat* × *'v clause option*

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

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

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

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

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

abbreviation *raw-backtrack-lvl* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'d* **where**
raw-backtrack-lvl ≡ λ(*M*, *N*, *U*, *k*, -). *k*

abbreviation *raw-update-backtrack-lvl* :: *'d* ⇒ *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e*
where
raw-update-backtrack-lvl ≡ λ*k* (*M*, *N*, *U*, -, *S*). (*M*, *N*, *U*, *k*, *S*)

abbreviation *raw-conflicting* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'e* **where**
raw-conflicting ≡ λ(*M*, *N*, *U*, *k*, *D*). *D*

abbreviation *raw-update-conflicting* :: *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e*
where
raw-update-conflicting ≡ λ*S* (*M*, *N*, *U*, *k*, -). (*M*, *N*, *U*, *k*, *S*)

abbreviation *S0-cdcl$_W$ N* ≡ (([], *N*, {#}, *0*, *None*):: *'v cdcl$_W$-state*)

abbreviation *raw-add-learned-clss* **where**
raw-add-learned-clss ≡ λ*C* (*M*, *N*, *U*, *S*). (*M*, *N*, {#*C*#} + *U*, *S*)

abbreviation *raw-remove-cls* **where**
raw-remove-cls ≡ λ*C* (*M*, *N*, *U*, *S*). (*M*, *removeAll-mset C N*, *removeAll-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*:
$S = (\textit{raw-trail}\ S,\ \textit{raw-init-clss}\ S,\ \textit{raw-learned-clss}\ S,\ \textit{raw-backtrack-lvl}\ S,\ \textit{raw-conflicting}\ S)$
⟨*proof*⟩

interpretation $\textit{state}_W$
raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting
$\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
$\lambda(M,\ S).\ (\textit{tl}\ M,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ \{\#C\#\} + U,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ \textit{removeAll-mset}\ C\ N,\ \textit{removeAll-mset}\ C\ U,\ S)$
$\lambda(k{::}\textit{nat})\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
$\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
$\lambda N.\ ([],\ N,\ \{\#\},\ 0,\ \textit{None})$
⟨*proof*⟩

interpretation $\textit{conflict-driven-clause-learning}_W$ *raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting*
$\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
$\lambda(M,\ S).\ (\textit{tl}\ M,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ \{\#C\#\} + U,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ \textit{removeAll-mset}\ C\ N,\ \textit{removeAll-mset}\ C\ U,\ S)$
$\lambda(k{::}\textit{nat})\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
$\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
$\lambda N.\ ([],\ N,\ \{\#\},\ 0,\ \textit{None})$
⟨*proof*⟩

declare *clauses-def*[*simp*]

lemma $\textit{cdcl}_W\text{-}\textit{state-eq-equality}$[*iff*]: $\textit{state-eq}\ S\ T \longleftrightarrow S = T$
⟨*proof*⟩
declare *state-simp*[*simp del*]

lemma *reduce-trail-to-empty-trail*[*simp*]:
$\textit{reduce-trail-to}\ F\ ([],\ aa,\ ab,\ ac,\ b) = ([],\ aa,\ ab,\ ac,\ b)$
⟨*proof*⟩

lemma *raw-trail-reduce-trail-to-length-le*:
assumes $\textit{length}\ F > \textit{length}\ (\textit{raw-trail}\ S)$
shows $\textit{raw-trail}\ (\textit{reduce-trail-to}\ F\ S) = []$
⟨*proof*⟩

lemma *reduce-trail-to*:
$\textit{reduce-trail-to}\ F\ S =$
$((\textit{if}\ \textit{length}\ (\textit{raw-trail}\ S) \geq \textit{length}\ F$
$\textit{then}\ \textit{drop}\ (\textit{length}\ (\textit{raw-trail}\ S) - \textit{length}\ F)\ (\textit{raw-trail}\ S)$
$\textit{else}\ []),\ \textit{raw-init-clss}\ S,\ \textit{raw-learned-clss}\ S,\ \textit{raw-backtrack-lvl}\ S,\ \textit{raw-conflicting}\ S)$
(**is** $?S = \text{-}$)
⟨*proof*⟩

7.1.5 CDCL Implementation

Definition of the rules

Types **lemma** *true-raw-init-clss-remdups*[*simp*]:
$I \models s\ (\textit{mset} \circ \textit{remdups})\ `\ N \longleftrightarrow\ I \models s\ \textit{mset}\ `\ N$

⟨*proof*⟩

lemma *satisfiable-mset-remdups*[*simp*]:
satisfiable ((*mset* ∘ *remdups*) ‘ *N*) ⟷ *satisfiable* (*mset* ‘ *N*)
⟨*proof*⟩

type-synonym *'v cdcl$_W$-state-inv-st* = (*'v*, *'v literal list*) *ann-lit list* ×
'v literal list list × *'v literal list list* × *nat* × *'v literal list option*

We need some functions to convert between our abstract state *'v cdcl$_W$-state* and the concrete state *'v cdcl$_W$-state-inv-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* ⟹ (∃ *C'*. *z* = *Propagated L C'* ∧ *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$_W$-state-inv-st* ⇒ *'v cdcl$_W$-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$_W$-state-inv* = {*S*::*'v cdcl$_W$-state-inv-st*. *cdcl$_W$-all-struct-inv* (*toS S*)}
morphisms *rough-state-of state-of*
⟨*proof*⟩

instantiation *cdcl$_W$-state-inv* :: (*type*) *equal*
begin
definition *equal-cdcl$_W$-state-inv* :: *'v cdcl$_W$-state-inv* ⇒ *'v cdcl$_W$-state-inv* ⇒ *bool* **where**
equal-cdcl$_W$-state-inv S S' = (*rough-state-of S* = *rough-state-of S'*)
instance
⟨*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* ⟷ *undefined-lit M L*
⟨*proof*⟩

lemma *true-annot-map-convert*[*simp*]: *map convert M* ⊨*a N* ⟷ *M* ⊨*a N*
⟨*proof*⟩

lemma *true-annots-map-convert*[*simp*]: *map convert M* ⊨*as N* ⟷ *M* ⊨*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 ≠ *S* ⟹ *propagate* (*toS S*) (*toS* (*do-propagate-step S*))
⟨*proof*⟩

lemma *do-propagate-step-option*[*simp*]:
raw-conflicting S ≠ *None* ⟹ *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* (∀ *c* ∈ *set N*. −*c* ∈ *lits-of-l M*) *then Some N else find-conflict M Ns*)

lemma *find-conflict-Some*:
find-conflict M Ns = *Some N* ⟹ *N* ∈ *set Ns* ∧ *M* ⊨*as CNot* (*mset N*)
⟨*proof*⟩

lemma *find-conflict-None*:
find-conflict M Ns = *None* ⟷ (∀ *N* ∈ *set Ns*. ¬*M* ⊨*as CNot* (*mset N*))
⟨*proof*⟩

lemma *find-conflict-None-no-confl*:
find-conflict M (*N*@*U*) = *None* ⟷ *no-step conflict* (*toS* (*M*, *N*, *U*, *k*, *None*))
⟨*proof*⟩

definition *do-conflict-step* **where**
do-conflict-step S =
(*case S of*
(*M*, *N*, *U*, *k*, *None*) ⇒
(*case find-conflict M* (*N* @ *U*) *of*
Some a ⇒ (*M*, *N*, *U*, *k*, *Some a*)
| *None* ⇒ (*M*, *N*, *U*, *k*, *None*))
| *S* ⇒ *S*)

lemma *do-conflict-step*:
do-conflict-step S ≠ *S* ⟹ *conflict* (*toS S*) (*toS* (*do-conflict-step S*))
⟨*proof*⟩

lemma *do-conflict-step-no-step*:
do-conflict-step S = *S* ⟹ *no-step conflict* (*toS S*)
⟨*proof*⟩

lemma *do-conflict-step-option*[*simp*]:
raw-conflicting S ≠ *None* ⟹ *do-conflict-step S* = *S*
⟨*proof*⟩

lemma *do-conflict-step-raw-conflicting*[*dest*]:
do-conflict-step S ≠ *S* ⟹ *raw-conflicting* (*do-conflict-step S*) ≠ *None*
⟨*proof*⟩

definition *do-cp-step* **where**
do-cp-step S =
(*do-propagate-step o do-conflict-step*) *S*

lemma *cp-step-is-cdcl*$_W$*-cp*:
assumes *H*: *do-cp-step S* ≠ *S*
shows *cdcl*$_W$*-cp* (*toS S*) (*toS* (*do-cp-step S*))
⟨*proof*⟩

lemma *do-cp-step-eq-no-prop-no-confl*:
do-cp-step S = *S* ⟹ *do-conflict-step S* = *S* ∧ *do-propagate-step S* = *S*
⟨*proof*⟩

lemma *no-cdcl*$_W$*-cp-iff-no-propagate-no-conflict*:
no-step cdcl$_W$*-cp S* ⟷ *no-step propagate S* ∧ *no-step conflict S*
⟨*proof*⟩

lemma *do-cp-step-eq-no-step*:
assumes *H*: *do-cp-step S* = *S* **and** ∀ *c* ∈ *set* (*raw-init-clss S* @ *raw-learned-clss S*). *distinct c*
shows *no-step cdcl*$_W$*-cp* (*toS S*)
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-cdcl*$_W$*-st*: *cdcl*$_W$*-cp S S*′ ⟹ *cdcl*$_W$** *S S*′
⟨*proof*⟩

lemma *cdcl*$_W$*-all-struct-inv-rough-state*[*simp*]: *cdcl*$_W$*-all-struct-inv* (*toS* (*rough-state-of S*))
⟨*proof*⟩

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* $-L \notin$ *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*) $\longleftrightarrow$ *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* $-L \in$ *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*) ⟷ *S* = (*a*, *b*, *c*, *d*, *None*)
⟨*proof*⟩

Backjumping **lemma** *get-all-ann-decomposition-map-convert*:
(*get-all-ann-decomposition* (*map convert M*)) =
map (λ(*a*, *b*). (*map convert a*, *map convert b*)) (*get-all-ann-decomposition M*)
⟨*proof*⟩

lemma *do-backtrack-step*:
assumes
db: *do-backtrack-step S* ≠ *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*′ ⟹ *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* ⟷ *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 ≠ *S* ⟹ *decide* (*toS S*) (*toS* (*do-decide-step S*))
⟨*proof*⟩

lemma *do-decide-step-no*:
do-decide-step S = S ⟹ no-step decide (toS S)
⟨*proof*⟩

lemma *rough-state-of-state-of-do-decide-step*[*simp*]:
cdcl$_W$-all-struct-inv (toS S) ⟹ rough-state-of (state-of (do-decide-step S)) = do-decide-step S
⟨*proof*⟩

lemma *rough-state-of-state-of-do-skip-step*[*simp*]:
cdcl$_W$-all-struct-inv (toS S) ⟹ rough-state-of (state-of (do-skip-step S)) = do-skip-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-of (if cdcl$_W$-all-struct-inv (toS (fst xs, snd xs)) then xs
else ([], [], [], 0, None))

lemma [*code abstype*]:
Con (rough-state-of S) = S
⟨*proof*⟩

definition *do-cp-step′* **where**
do-cp-step′ S = state-of (do-cp-step (rough-state-of S))

typedef *'v cdcl$_W$-state-inv-from-init-state =*
{S:: 'v cdcl$_W$-state-inv-st. cdcl$_W$-all-struct-inv (toS S)
∧ cdcl$_W$-stgy** (S0-cdcl$_W$ (raw-init-clss (toS S))) (toS S)}
morphisms *rough-state-from-init-state-of state-from-init-state-of*
⟨*proof*⟩

instantiation *cdcl$_W$-state-inv-from-init-state :: (type) equal*
begin
definition *equal-cdcl$_W$-state-inv-from-init-state :: 'v cdcl$_W$-state-inv-from-init-state ⇒*
'v cdcl$_W$-state-inv-from-init-state ⇒ bool **where**
equal-cdcl$_W$-state-inv-from-init-state S S′ ⟷
(rough-state-from-init-state-of S = rough-state-from-init-state-of S′)
instance
⟨*proof*⟩
end

definition *ConI* **where**
ConI S = state-from-init-state-of (if cdcl$_W$-all-struct-inv (toS (fst S, snd S))
∧ cdcl$_W$-stgy** (S0-cdcl$_W$ (raw-init-clss (toS S))) (toS S) then S else ([], [], [], 0, None))

lemma [*code abstype*]:
ConI (rough-state-from-init-state-of S) = S
⟨*proof*⟩

definition *id-of-I-to*:: *'v cdcl$_W$-state-inv-from-init-state ⇒ 'v cdcl$_W$-state-inv* **where**

id-of-I-to S = state-of (rough-state-from-init-state-of S)

lemma [*code abstract*]:
rough-state-of (id-of-I-to S) = rough-state-from-init-state-of S
⟨*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 ≠ S
then T
else
(let U = do-resolve-step T in
if U ≠ T
then U else
(let V = do-backtrack-step U in
if V ≠ U then V else do-decide-step V)))

lemma *do-other-step*:
assumes *inv*: *cdcl$_W$-all-struct-inv (toS S)* **and**
st: *do-other-step S ≠ 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*:
$\exists\, c.\ raw\text{-}trail\ (do\text{-}cp\text{-}step\ S) = c\ @\ raw\text{-}trail\ \ S \wedge (\forall\, m \in set\ c.\ \neg\ is\text{-}decided\ m)$
⟨*proof*⟩

lemma *do-full1-cp-step-neq-raw-trail-increase*:
$\exists\, c.\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S)) = c\ @\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)$
$\wedge\ (\forall\, m \in set\ c.\ \neg\ is\text{-}decided\ m)$
⟨*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*]:
$rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}skip\text{-}step\ (rough\text{-}state\text{-}of\ S))) = do\text{-}skip\text{-}step\ (rough\text{-}state\text{-}of\ S)$

⟨*proof*⟩

lemma *raw-conflicting-do-resolve-step-iff*[*iff*]:
raw-conflicting (*do-resolve-step S*) = *None* ⟷ *raw-conflicting S* = *None*
⟨*proof*⟩

lemma *raw-conflicting-do-skip-step-iff*[*iff*]:
raw-conflicting (*do-skip-step S*) = *None* ⟷ *raw-conflicting S* = *None*
⟨*proof*⟩

lemma *raw-conflicting-do-decide-step-iff*[*iff*]:
raw-conflicting (*do-decide-step S*) = *None* ⟷ *raw-conflicting S* = *None*
⟨*proof*⟩

lemma *raw-conflicting-do-backtrack-step-imp*[*simp*]:
do-backtrack-step S ≠ *S* ⟹ *raw-conflicting* (*do-backtrack-step S*) = *None*
⟨*proof*⟩

lemma *do-skip-step-eq-iff-raw-trail-eq*:
do-skip-step S = *S* ⟷ *raw-trail* (*do-skip-step S*) = *raw-trail S*
⟨*proof*⟩

lemma *do-decide-step-eq-iff-raw-trail-eq*:
do-decide-step S = *S* ⟷ *raw-trail* (*do-decide-step S*) = *raw-trail S*
⟨*proof*⟩

lemma *do-backtrack-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail S*)
shows *do-backtrack-step S* = *S* ⟷ *raw-trail* (*do-backtrack-step S*) = *raw-trail S*
⟨*proof*⟩

lemma *do-resolve-step-eq-iff-raw-trail-eq*:
do-resolve-step S = *S* ⟷ *raw-trail* (*do-resolve-step S*) = *raw-trail S*
⟨*proof*⟩

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* ⟷ *do-other-step S* = *S*
⟨*proof*⟩

lemma *do-full1-cp-step-do-other-step′-normal-form*[*dest*!]:
assumes *H*: *do-full1-cp-step* (*do-other-step′ S*) = *S*
shows *do-other-step′ S* = *S* ∧ *do-full1-cp-step S* = *S*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma $cdcl_W$*-cp-is-rtranclp-cdcl*$_W$: $cdcl_W$*-cp S T* $\Longrightarrow$ $cdcl_W^{**}$ *S T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-is-rtranclp-cdcl*$_W$: $cdcl_W$*-cp*** *S T* $\Longrightarrow$ $cdcl_W^{**}$ *S T*
⟨*proof*⟩

lemma $cdcl_W$*-stgy-is-rtranclp-cdcl*$_W$:
$cdcl_W$*-stgy S T* $\Longrightarrow$ $cdcl_W^{**}$ *S T*
⟨*proof*⟩

lemma $cdcl_W$*-stgy-init-raw-init-clss*:
$cdcl_W$*-stgy S T* $\Longrightarrow$ $cdcl_W$*-M-level-inv S* $\Longrightarrow$ *raw-init-clss S* = *raw-init-clss T*
⟨*proof*⟩

lemma *clauses-toS-rough-state-of-do-cdcl*$_W$*-stgy-step*[*simp*]:
raw-init-clss (*toS* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*state-of* (*rough-state-from-init-state-of S*))))) = *raw-init-clss* (*toS* (*rough-state-from-init-state-of S*)) (**is** - = *raw-init-clss* (*toS ?S*))
⟨*proof*⟩

lemma *rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step'*[*code abstract*]:
rough-state-from-init-state-of (*do-cdcl*$_W$*-stgy-step'* *S*) =
rough-state-of (*do-cdcl*$_W$*-stgy-step* (*id-of-I-to S*))
⟨*proof*⟩

All rules together **function** *do-all-cdcl*$_W$*-stgy* **where**
do-all-cdcl$_W$*-stgy S* =
(*let T* = *do-cdcl*$_W$*-stgy-step' S in*
if T = *S then S else do-all-cdcl*$_W$*-stgy T*)
⟨*proof*⟩
termination
⟨*proof*⟩

thm *do-all-cdcl*$_W$*-stgy.induct*
lemma *do-all-cdcl*$_W$*-stgy-induct*:
(⋀*S*. (*do-cdcl*$_W$*-stgy-step' S* ≠ *S* $\Longrightarrow$ *P* (*do-cdcl*$_W$*-stgy-step' S*)) $\Longrightarrow$ *P S*) $\Longrightarrow$ *P a0*
⟨*proof*⟩

lemma *no-step-cdcl*$_W$*-stgy-cdcl*$_W$*-all*:
fixes *S* :: *'a cdcl*$_W$*-state-inv-from-init-state*
shows *no-step cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy S*)))
⟨*proof*⟩

lemma *do-all-cdcl*$_W$*-stgy-is-rtranclp-cdcl*$_W$*-stgy*:
$cdcl_W$*-stgy*** (*toS* (*rough-state-from-init-state-of S*))
(*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy S*)))
⟨*proof*⟩

Final theorem:

lemma *DPLL-tot-correct*:
assumes
r: *rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy* (*state-from-init-state-of*
((], *map remdups N*, [], *0*, *None*)))) = *S* **and**
S: (*M'*, *N'*, *U'*, *k*, *E*) = *toS S*
shows (*E* ≠ *Some* {#} ∧ *satisfiable* (*set* (*map mset N*)))
∨ (*E* = *Some* {#} ∧ *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* ⇒ *'v clause*
begin
end

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

context
begin
  interpretation *list-cls*: *raw-cls mset*
    ⟨*proof*⟩

  interpretation *cls-cls*: *raw-cls id*
    ⟨*proof*⟩

end

Over the abstract clauses, we have the following properties:

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

locale *raw-clss* =
  raw-cls mset-cls
  for
    mset-cls :: *'cls* ⇒ *'v clause* +

fixes
mset-clss:: *'clss* ⇒ *'v clauses* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss*
assumes
insert-clss[*simp*]: *mset-clss* (*insert-clss L C*) = *mset-clss C* + {#*mset-cls L*#} **and**
in-clss-mset-clss[*dest*]: *in-clss a C* ⟹ *mset-cls a* ∈# *mset-clss C* **and**
in-mset-clss-exists-preimage: *b* ∈# *mset-clss C* ⟹ ∃ *b'*. *in-clss b' C* ∧ *mset-cls b'* = *b*
begin

end

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

lemma *mset-map-mset-remove-first*:
mset (*map mset* (*remove-first a C*)) = *remove1-mset* (*mset a*) (*mset* (*map mset C*))
⟨*proof*⟩

interpretation *clss-clss*: *raw-clss id*
id op ∈# *λL C*. *C* + {#*L*#}
⟨*proof*⟩

interpretation *list-clss*: *raw-clss mset*
λL. *mset* (*map mset L*) *λL C*. *L* ∈ *set C op* #
⟨*proof*⟩
end

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

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*$_W$*-mset* = (*'v*, *'v clause*) *ann-lit list* ×
'v clauses ×
'v clauses ×
nat × *'v clause option*

We use definition, otherwise we could not use the simplification theorems we have already shown.

definition *trail* :: *'v cdcl*$_W$*-mset* ⇒ (*'v*, *'v clause*) *ann-lit list* **where**
trail ≡ *λ*(*M*, -). *M*

definition *init-clss* :: *'v cdcl$_W$-mset* ⇒ *'v clauses* **where**
init-clss ≡ λ(-, *N*, -). *N*

definition *learned-clss* :: *'v cdcl$_W$-mset* ⇒ *'v clauses* **where**
learned-clss ≡ λ(-, -, *U*, -). *U*

definition *backtrack-lvl* :: *'v cdcl$_W$-mset* ⇒ *nat* **where**
backtrack-lvl ≡ λ(-, -, -, *k*, -). *k*

definition *conflicting* :: *'v cdcl$_W$-mset* ⇒ *'v clause option* **where**
conflicting ≡ λ(-, -, -, -, *C*). *C*

definition *cons-trail* :: (*'v*, *'v clause*) *ann-lit* ⇒ *'v cdcl$_W$-mset* ⇒ *'v cdcl$_W$-mset* **where**
cons-trail ≡ λ*L* (*M*, *R*). (*L* # *M*, *R*)

definition *tl-trail* **where**
tl-trail ≡ λ(*M*, *R*). (*tl M*, *R*)

definition *add-learned-cls* **where**
add-learned-cls ≡ λ*C* (*M*, *N*, *U*, *R*). (*M*, *N*, {#*C*#} + *U*, *R*)

definition *remove-cls* **where**
remove-cls ≡ λ*C* (*M*, *N*, *U*, *R*). (*M*, *removeAll-mset C N*, *removeAll-mset C U*, *R*)

definition *update-backtrack-lvl* **where**
update-backtrack-lvl ≡ λ*k* (*M*, *N*, *U*, -, *D*). (*M*, *N*, *U*, *k*, *D*)

definition *update-conflicting* **where**
update-conflicting ≡ λ*D* (*M*, *N*, *U*, *k*, -). (*M*, *N*, *U*, *k*, *D*)

definition *init-state* **where**
init-state ≡ λ*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*
⟨*proof*⟩

lemma $cdcl_W$*-mset-state-eq-eq*: $cdcl_W$*-mset.state-eq* = (*op* =)
⟨*proof*⟩

notation $cdcl_W$*-mset.state-eq* (**infix** $\sim m$ *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* $\sim m$ *state S'* **and**
TT': *state T* $\sim m$ *state T'*
shows *R-abs S' T'*
⟨*proof*⟩

lemma *rtranclp-relation-invariant*:
R^{++} *S T* $\Longrightarrow$ *inv S* $\Longrightarrow$ *inv T*
⟨*proof*⟩

lemma *rtranclp-abs-rtranclp*:
R-abs** *S T* $\Longrightarrow$ *inv* (*state S*) $\Longrightarrow$ R^{**} (*state S*) (*state T*)
⟨*proof*⟩

lemma *tranclp-relation-tranclp-relation-abs-compatible*:
fixes *S* :: *'st*
assumes
R: R^{++} (*state S*) *T* **and**
inv: *inv* (*state S*)
shows $\exists$ *U*. *R-abs*$^{++}$ *S U* $\wedge$ *T* $\sim m$ *state U*
⟨*proof*⟩

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-abs*** *S U* $\wedge$ *T* $\sim m$ *state U*
⟨*proof*⟩

lemma *no-step-iff*:
inv (*state S*) $\Longrightarrow$ *no-step R* (*state S*) $\longleftrightarrow$ *no-step R-abs S*
⟨*proof*⟩

lemma *tranclp-relation-compatible-eq-and-inv*:
assumes
inv: *inv* (*state S*) **and**
st: *R-abs*$^{++}$ *S T* **and**
SS': *state S* $\sim m$ *state S'* **and**
TU: *state T* $\sim m$ *state U*
shows *R-abs*$^{++}$ *S' U* $\wedge$ *inv* (*state U*)
⟨*proof*⟩

lemma
assumes
inv: *inv* (*state S*) **and**
st: *R-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-abs*$^{++}$ *S' U* **and**

tranclp-relation-abs-invariant: *inv* (*state U*)
⟨*proof*⟩

lemma *tranclp-abs-tranclp*: *R-abs*$^{++}$ *S T* ⟹ *inv* (*state S*) ⟹ *R*$^{++}$ (*state S*) (*state T*)
⟨*proof*⟩

lemma *full1-iff*:
assumes *inv*: *inv* (*state S*)
shows *full1 R* (*state S*) (*state T*) ⟷ *full1 R-abs S T* (**is** *?R* ⟷ *?R-abs*)
⟨*proof*⟩

lemma *full1-iff-compatible*:
assumes *inv*: *inv* (*state S*) **and** *SS'*: *S'* ∼*m state S* **and** *TT'*: *T'* ∼*m state T*
shows *full1 R S' T'* ⟷ *full1 R-abs S T* (**is** *?R* ⟷ *?R-abs*)
⟨*proof*⟩

lemma *full-if-full-abs*:
assumes *inv* (*state S*) **and** *full R-abs S T*
shows *full R* (*state S*) (*state T*)
⟨*proof*⟩

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-abs S T* ∨ (*state S* ∼*m state T* ∧ *no-step R* (*state S*))
⟨*proof*⟩

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* ∼*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* ∼*m state U*
⟨*proof*⟩

lemma *full1-right-compatible*:
assumes *inv* (*state S*) **and**
full1: *full1 R-abs S T* **and** *TV*: *state T* ∼*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* ∼*m state U*
shows *full R-abs S U* ∨ (*S* = *T* ∧ *no-step R-abs S*)
⟨*proof*⟩

end

locale *relation-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* (*state S*) (*state T*) ⟷ *R-abs S 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*
begin

lemma *relation-compatible-eq*:
inv (*state S*) ⟹ *R-abs S T* ⟹ *state S* ∼*m state S'* ⟹ *state T* ∼*m state T'* ⟹ *R-abs S' T'*
⟨*proof*⟩

lemma *relation-right-compatible*:
inv (*state S*) ⟹ *R-abs S T* ⟹ *state T* ∼*m state U* ⟹ *R-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$_W$-ops* =
raw-clss mset-cls
mset-clss in-clss insert-clss
+
raw-cls mset-ccls
for
— Clause
mset-cls :: *'cls* ⇒ *'v clause* **and**

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

mset-ccls :: *'ccls* ⇒ *'v clause*
+
fixes
ccls-of-cls :: *'cls* ⇒ *'ccls* **and**
cls-of-ccls :: *'ccls* ⇒ *'cls* **and**

conc-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-conc-trail :: *'st* ⇒ (*'v*, *'cls*) *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*) *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 :: *'ccls* ⇒ *'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*
assumes
mset-ccls-ccls-of-cls[*simp*]:
mset-ccls (*ccls-of-cls C*) = *mset-cls C* **and**
mset-cls-cls-of-ccls[*simp*]:
mset-cls (*cls-of-ccls D*) = *mset-ccls D* **and**
ex-mset-cls: ∃ *a*. *mset-cls a* = *E*
begin

fun *mmset-of-mlit* :: (*'v*, *'cls*) *ann-lit* ⇒ (*'v*, *'v clause*) *ann-lit*
where
mmset-of-mlit (*Propagated L C*) = *Propagated L* (*mset-cls C*) |
mmset-of-mlit (*Decided L*) = *Decided L*

lemma *lit-of-mmset-of-mlit*[*simp*]:
lit-of (*mmset-of-mlit a*) = *lit-of a*
⟨*proof*⟩

lemma *lit-of-mmset-of-mlit-set-lit-of-l*[*simp*]:
lit-of ‘ *mmset-of-mlit* ‘ *set M'* = *lits-of-l M'*
⟨*proof*⟩

lemma *map-mmset-of-mlit-true-annots-true-cls*[*simp*]:
map mmset-of-mlit M' ⊨*as C* ⟷ *M'* ⊨*as C*
⟨*proof*⟩

definition *conc-init-clss* ≡ λ*S*. *mset-clss* (*raw-clauses S*) − *conc-learned-clss S*
abbreviation *conc-conflicting* ≡ λ*S*. *map-option mset-ccls* (*raw-conc-conflicting S*)

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

abbreviation *conc-clauses* :: *'st* ⇒ *'v clauses* **where**
conc-clauses S ≡ *mset-clss* (*raw-clauses 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))

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 $op \cup$ operator.

locale *abs-state$_W$* =
  abs-state$_W$-ops
    — functions for clauses:
    mset-cls
      mset-clss in-clss insert-clss

    — functions for the conflicting clause:
    mset-ccls

    — Conversion between conflicting and non-conflicting
    ccls-of-cls cls-of-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
mset-cls :: *'cls* ⇒ *'v clause* **and**

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

mset-ccls :: *'ccls* ⇒ *'v clause* **and**

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

conc-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-conc-trail :: *'st* ⇒ (*'v*, *'cls*) *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*) *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 :: *'ccls* ⇒ *'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* +
assumes
— Definition of *hd-raw-trail*:
hd-raw-conc-trail:
conc-trail S ≠ [] ⟹ *mmset-of-mlit* (*hd-raw-conc-trail S*) = *hd* (*conc-trail S*) **and**

cons-conc-trail:
⋀*S'*. *undefined-lit* (*conc-trail st*) (*lit-of L*) ⟹
state st = (*M*, *S'*) ⟹
state (*cons-conc-trail L st*) = (*mmset-of-mlit L* # *M*, *S'*) **and**

tl-conc-trail:
⋀*S'*. *state st* = (*M*, *S'*) ⟹ *state* (*tl-conc-trail st*) = (*tl M*, *S'*) **and**

remove-cls:
⋀*S'*. *state st* = (*M*, *N*, *U*, *S'*) ⟹
state (*remove-cls C st*) =
(*M*, *removeAll-mset* (*mset-cls C*) *N*, *removeAll-mset* (*mset-cls C*) *U*, *S'*) **and**

add-conc-confl-to-learned-cls:
no-dup (*conc-trail st*) ⟹ *state st* = (*M*, *N*, *U*, *k*, *Some F*) ⟹

state (*add-conc-confl-to-learned-cls st*) =
(*M*, *N*, {#*F*#} + *U*, *k*, *None*) **and**

update-conc-backtrack-lvl:
⋀*S'*. *state st* = (*M*, *N*, *U*, *k*, *S'*) ⟹
state (*update-conc-backtrack-lvl k' st*) = (*M*, *N*, *U*, *k'*, *S'*) **and**

mark-conflicting:
state st = (*M*, *N*, *U*, *k*, *None*) ⟹
state (*mark-conflicting E st*) = (*M*, *N*, *U*, *k*, *Some* (*mset-ccls E*)) **and**

conc-conflicting-mark-conflicting[*simp*]:
raw-conc-conflicting (*mark-conflicting E st*) = *Some E* **and**

resolve-conflicting:
state st = (*M*, *N*, *U*, *k*, *Some F*) ⟹ −*L'* ∈# *F* ⟹ *L'* ∈# *mset-cls D* ⟹
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*) = ([], *mset-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*) ⊆# *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*]:
⋀*S'*. *conc-trail st* = *M2* @ *M1* ⟹ *state st* = (*M*, *S'*) ⟹
state (*reduce-conc-trail-to M1 st*) = (*M1*, *S'*) **and**

learned-clauses:
conc-learned-clss S ⊆# *conc-clauses S*
begin

lemma
— Properties about the trail *conc-trail*:
conc-trail-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) ⟹
conc-trail (*cons-conc-trail L st*) = *mmset-of-mlit 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*) ⟹ *conc-conflicting st* ≠ *None* ⟹
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* ⟹ *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*) $\Longrightarrow$
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*) $\Longrightarrow$
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* $\Longrightarrow$
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*) $\Longrightarrow$
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* $\Longrightarrow$
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*) $\Longrightarrow$
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*) = *mset-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*⟩

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* $\Longrightarrow$
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*⟩

abbreviation *incr-lvl* :: *'st* ⇒ *'st* **where**
incr-lvl S ≡ *update-conc-backtrack-lvl* (*conc-backtrack-lvl S* + *1*) *S*

abbreviation *state-eq* :: *'st* ⇒ *'st* ⇒ *bool* (**infix** ∼ *36*) **where**
S ∼ *T* ≡ *state S* ∼*m state T*

lemma *state-eq-sym*:
S ∼ *T* ⟷ *T* ∼ *S*
⟨*proof*⟩

lemma *state-eq-trans*:
S ∼ *T* ⟹ *T* ∼ *U* ⟹ *S* ∼ *U*
⟨*proof*⟩

lemma *conc-clauses-init-learned*: *conc-clauses S* = *conc-init-clss S* + *conc-learned-clss S*
⟨*proof*⟩

lemma
shows
state-eq-conc-trail: *S* ∼ *T* ⟹ *conc-trail S* = *conc-trail T* **and**
state-eq-conc-init-clss: *S* ∼ *T* ⟹ *conc-init-clss S* = *conc-init-clss T* **and**
state-eq-conc-learned-clss: *S* ∼ *T* ⟹ *conc-learned-clss S* = *conc-learned-clss T* **and**
state-eq-conc-backtrack-lvl: *S* ∼ *T* ⟹ *conc-backtrack-lvl S* = *conc-backtrack-lvl T* **and**
state-eq-conc-conflicting: *S* ∼ *T* ⟹ *conc-conflicting S* = *conc-conflicting T* **and**
state-eq-clauses: *S* ∼ *T* ⟹ *conc-clauses S* = *conc-clauses T* **and**
state-eq-undefined-lit:
S ∼ *T* ⟹ *undefined-lit* (*conc-trail S*) *L* = *undefined-lit* (*conc-trail T*) *L*
⟨*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* = *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 ∈ *atms-of-mm* (*conc-learned-clss* (*restart-state S*)) ⟹ *x* ∈ *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-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*)
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*⟩

end — end of $state_W$ locale

7.2.4 CDCL Rules

locale $abs\text{-}conflict\text{-}driven\text{-}clause\text{-}learning_W$ =
$abs\text{-}state_W$
— functions for clauses:
mset-cls
mset-clss in-clss insert-clss

— functions for the conflicting clause:
mset-ccls

— conversion
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss
— changing state:
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

— get state:
conc-init-state
restart-state
for
mset-cls :: *'cls* ⇒ *'v clause* **and**

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

mset-ccls :: *'ccls* ⇒ *'v clause* **and**

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

conc-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-conc-trail :: *'st* ⇒ (*'v*, *'cls*) *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*) *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 :: *'ccls* ⇒ *'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

lemma *clauses-state-conc-clauses*[*simp*]: $cdcl_W$*-mset.clauses* (*state S*) = *conc-clauses S*
⟨*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* ⟹ *trail* (*state* (*mark-conflicting i S*)) = *trail* (*state S*)
⟨*proof*⟩

lemma *trail-state-conc-trail*[*simp*]:
trail (*state S*) = *conc-trail S*
⟨*proof*⟩

lemma *init-clss-state-conc-init-clss*[*simp*]:
init-clss (*state S*) = *conc-init-clss S*
⟨*proof*⟩

lemma *learned-clss-state-conc-learned-clss*[*simp*]:
learned-clss (*state S*) = *conc-learned-clss 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*) ⟹
cons-trail (*mmset-of-mlit 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* ⟹
cons-trail (*Propagated K* (*mset-cls C*)) (*state S*) = *state* (*cons-conc-trail* (*Propagated K C*) *S*)
⟨*proof*⟩

lemma *state-cons-cons-trail-cons-trail-propagated-ccls*[*simp*]:
undefined-lit (*trail* (*state S*)) *K* ⟹
cons-trail (*Propagated K* (*mset-ccls C*)) (*state S*) =
state (*cons-conc-trail* (*Propagated K* (*cls-of-ccls C*)) *S*)
⟨*proof*⟩

lemma *state-cons-cons-trail-cons-trail-decided*[*simp*]:
undefined-lit (*trail* (*state S*)) *K* ⟹
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*
shows
update-conflicting (*Some* (*mset-ccls D*)) (*state S*) = *state* (*mark-conflicting D S*)
update-conflicting (*Some* (*mset-cls D′*)) (*state S*) =
state (*mark-conflicting* ((*ccls-of-cls D′*)) *S*)
⟨*proof*⟩

lemma *update-backtrack-lvl-state*[*simp*]:
update-backtrack-lvl i (*state S*) = *state* (*update-conc-backtrack-lvl i S*)
⟨*proof*⟩

lemma *conc-conflicting-conflicting*[*simp*]:
conflicting (*state S*) = *conc-conflicting S*
⟨*proof*⟩

lemma *update-conflicting-resolve-state-mark-conflicting*[*simp*]:
raw-conc-conflicting S = *Some D′* ⟹ −*L* ∈# *mset-ccls D′* ⟹ *L* ∈# *mset-cls E′* ⟹
update-conflicting (*Some* (*remove1-mset* (− *L*) (*mset-ccls D′*) #∪ *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*) ⟹ *raw-conc-conflicting S* = *Some D′* ⟹ *add-learned-cls* (*mset-ccls 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 *conc-backtrack-lvl-backtrack-lvl*[*simp*]:
backtrack-lvl (*state S*) = *conc-backtrack-lvl 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 (mset-clss N)*
⟨*proof*⟩

More robust version of *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 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*⟩

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

inductive-cases *propagate-absE*: *propagate-abs S T*

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 ⟹
D !∈! raw-clauses S ⟹
conc-trail S ⊨as CNot (mset-cls D) ⟹
T ∼ mark-conflicting (ccls-of-cls D) S ⟹
conflict-abs S T

inductive-cases *conflict-absE*: *conflict-abs S T*

lemma *conflict-conflict-abs*:
cdcl$_W$-mset.conflict (state S) (state T) ⟷ conflict-abs S T (**is** *?mset ⟷ ?abs*)
⟨*proof*⟩

lemma *conflict-compatible-abs*:
  assumes *SS'*: *S* ∼*m state S'* **and** *conflict*: *cdcl*$_W$*-mset.conflict S T*
  obtains *U* **where** *conflict-abs S' U* **and** *T* ∼*m state U*
⟨*proof*⟩

interpretation *conflict-abs*: *relation-relation-abs cdcl*$_W$*-mset.conflict conflict-abs state*
  λ-. True
  ⟨*proof*⟩

inductive *backtrack-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
backtrack-abs-rule:
  raw-conc-conflicting S = *Some D* ⟹
  L ∈# *mset-ccls D* ⟹
  (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail S*)) ⟹
  get-level (*conc-trail S*) *L* = *conc-backtrack-lvl S* ⟹
  get-level (*conc-trail S*) *L* = *get-maximum-level* (*conc-trail S*) (*mset-ccls D*) ⟹
  get-maximum-level (*conc-trail S*) (*mset-ccls D* − {#*L*#}) ≡ *i* ⟹
  get-level (*conc-trail S*) *K* = *i* + *1* ⟹
  T ∼ *cons-conc-trail* (*Propagated L* (*cls-of-ccls D*))
      (*reduce-conc-trail-to M1*
        (*add-conc-confl-to-learned-cls*
          (*update-conc-backtrack-lvl i S*))) ⟹
  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*) ⟷ *backtrack-abs S T* (**is** *?conc* ⟷ *?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* ∼*m state S'*
  obtains *U* **where** *backtrack-abs S' U* **and** *T* ∼*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* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
decide-abs-rule:
  conc-conflicting S = *None* ⟹
  undefined-lit (*conc-trail S*) *L* ⟹
  atm-of L ∈ *atms-of-mm* (*conc-init-clss S*) ⟹
  T ∼ *cons-conc-trail* (*Decided L*) (*incr-lvl S*) ⟹
  decide-abs S T

inductive-cases *decide-absE*: *decide-abs S T*

lemma *decide-decide-abs*:
  cdcl$_W$*-mset.decide* (*state S*) (*state T*) ⟷ *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 ⇒ 'st ⇒ bool* **for** *S* :: *'st* **where**
skip-abs-rule:
  conc-trail S = Propagated L C' # M ⟹
  raw-conc-conflicting S = Some E ⟹
  −L ∉# mset-ccls E ⟹
  mset-ccls E ≠ {#} ⟹
  T ∼ tl-conc-trail S ⟹
  skip-abs S T

inductive-cases *skip-absE*: *skip-abs S T*

lemma *skip-skip-abs*:
  cdcl$_W$-mset.skip (*state S*) (*state T*) ⟷ *skip-abs S T* (**is** *?conc* ⟷ *?abs*)
⟨*proof*⟩

lemma *skip-exists-skip-abs*:
  assumes *skip*: *cdcl$_W$-mset.skip S T* **and** *SS'*: *S ∼m state S'*
  obtains *U* **where** *skip-abs S' U* **and** *T ∼m state U*
⟨*proof*⟩

interpretation *skip-abs*: *relation-relation-abs cdcl$_W$-mset.skip skip-abs state*
  λ-. True
  ⟨*proof*⟩

inductive *resolve-abs* :: *'st ⇒ 'st ⇒ bool* **for** *S* :: *'st* **where**
resolve-abs-rule: *conc-trail S ≠ [] ⟹*
  hd-raw-conc-trail S = Propagated L E ⟹
  L ∈# mset-cls E ⟹
  raw-conc-conflicting S = Some D' ⟹
  −L ∈# mset-ccls D' ⟹
  get-maximum-level (*conc-trail S*) (*remove1-mset* (*−L*) (*mset-ccls D'*)) = *conc-backtrack-lvl S ⟹*
  T ∼ resolve-conflicting L E (*tl-conc-trail S*) ⟹
  resolve-abs S T

inductive-cases *resolve-absE*: *resolve-abs S T*

lemma *resolve-resolve-abs*:
  cdcl$_W$-mset.resolve (*state S*) (*state T*) ⟷ *resolve-abs S T* (**is** *?conc* ⟷ *?abs*)
⟨*proof*⟩

lemma *resolve-exists-resolve-abs*:
  assumes
    res: *cdcl$_W$-mset.resolve S T* **and**
    SS': *S ∼m state S'*
  obtains *U* **where** *resolve-abs S' U* **and** *T ∼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* $\Longrightarrow$
  $\neg$*conc-trail S* $\models$*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* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **where**
forget-rule:
  conc-conflicting S = None $\Longrightarrow$
  C !$\in$! *raw-conc-learned-clss S* $\Longrightarrow$
  $\neg$(*conc-trail S*) $\models$*asm clauses S* $\Longrightarrow$
  mset-cls C $\notin$ *set* (*get-all-mark-of-propagated* (*conc-trail S*)) $\Longrightarrow$
  mset-cls C $\notin\#$ *conc-init-clss S* $\Longrightarrow$
  T $\sim$ *remove-cls C S* $\Longrightarrow$
  forget S T

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

inductive $cdcl_W$*-abs-rf* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *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* $\Rightarrow$ *'st* $\Rightarrow$ *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* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
decide: *decide-abs S S'* $\Longrightarrow$ $cdcl_W$*-abs-o S S'* |
bj: $cdcl_W$*-abs-bj S S'* $\Longrightarrow$ $cdcl_W$*-abs-o S S'*

inductive $cdcl_W$*-abs* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
propagate: *propagate-abs S S'* $\Longrightarrow$ $cdcl_W$*-abs S S'* |
conflict: *conflict-abs S S'* $\Longrightarrow$ $cdcl_W$*-abs S S'* |
other: $cdcl_W$*-abs-o S S'* $\Longrightarrow$ $cdcl_W$*-abs S S'*|
rf: $cdcl_W$*-abs-rf S S'* $\Longrightarrow$ $cdcl_W$*-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* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**

conflict': *conflict-abs S T* $\Longrightarrow$ *full cdcl$_W$-abs-bj T U* $\Longrightarrow$ *cdcl$_W$-merge-abs-cp S U* |
propagate': *propagate-abs*$^{++}$ *S S'* $\Longrightarrow$ *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* $\sim m$ *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 $\Longrightarrow$ *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*) $\Longrightarrow$
  V $\sim$ *W* $\Longrightarrow$ *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* ∼ *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* (*mset-clss N*)
shows (*conc-conflicting T* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*mset-clss N*)))
∨ (*conc-conflicting T* = *None* ∧ *conc-trail T* ⊨*asm mset-clss N*
∧ *satisfiable* (*set-mset* (*mset-clss N*)))
⟨*proof*⟩

end

end

7.3 2-Watched-Literal

theory *CDCL-Two-Watched-Literals*
imports *CDCL-W-Abstract-State*
begin

First we define here the core of the two-watched literal data structure:

1. A clause is composed of (at most) two watched literals.
2. It is sufficient to find the candidates for propagation and conflict from the clauses such that the new literal is watched.

While this it the principle behind the two-watched literals, an implementation have to remember the candidates that have been found so far while updating the data structure.

We will directly on the two-watched literals data structure with lists: it could be also seen as a state over some abstract clause representation we would later refine as lists. However, as we need a way to select element from a clause, working on lists is better.

7.3.1 Essence of 2-WL

Data structure and Access Functions

Only the 2-watched literals have to be verified here: the backtrack level and the trail that appear in the state are not related to the 2-watched algoritm.

datatype *'v twl-clause* =
TWL-Clause (*watched*: *'v literal list*) (*unwatched*: *'v literal list*)

datatype *'v twl-state* =
TWL-State (*raw-trail*: (*'v*, *'v twl-clause*) *ann-lits*)

(*raw-init-clss*: *'v twl-clause list*)
(*raw-learned-clss*: *'v twl-clause list*) (*backtrack-lvl*: *nat*)
(*raw-conflicting*: *'v literal list option*)

fun *mmset-of-mlit* :: (*'v*, *'v twl-clause*) *ann-lit* ⇒ (*'v*, *'v clause*) *ann-lit*
where
mmset-of-mlit (*Propagated L C*) = *Propagated L* (*mset* (*watched C* @ *unwatched C*)) |
mmset-of-mlit (*Decided L*) = *Decided L*

lemma *lit-of-mmset-of-mlit*[*simp*]: *lit-of* (*mmset-of-mlit x*) = *lit-of x*
⟨*proof*⟩

lemma *lits-of-mmset-of-mlit*[*simp*]: *lits-of* (*mmset-of-mlit* ‘ *S*) = *lits-of S*
⟨*proof*⟩

abbreviation *trail* **where**
trail S ≡ *map mmset-of-mlit* (*raw-trail S*)

abbreviation *clauses-of-l* **where**
clauses-of-l ≡ λ*L*. *mset* (*map mset L*)

definition *raw-clause* :: *'v twl-clause* ⇒ *'v literal list* **where**
raw-clause C ≡ *watched C* @ *unwatched C*

definition *clause* :: *'v twl-clause* ⇒ *'v clause* **where**
clause C ≡ *mset* (*raw-clause C*)

lemma *clause-def-lambda*:
clause = (λ*C*. *mset* (*raw-clause C*))
⟨*proof*⟩

abbreviation *raw-clss-l* :: *'a twl-clause list* ⇒ *'a clauses* **where**
raw-clss-l C ≡ *mset* (*map clause C*)

abbreviation *raw-clauses* :: *'v twl-state* ⇒ *'v twl-clause list* **where**
raw-clauses S ≡ *raw-init-clss S* @ *raw-learned-clss S*

abbreviation *raw-clss* :: *'v twl-state* ⇒ *'v clauses* **where**
raw-clss S ≡ *raw-clss-l* (*raw-clauses S*)

interpretation *raw-cls clause* ⟨*proof*⟩

lemma *mset-map-clause-remove1-cond*:
raw-clss-l (*remove1-cond* (λ*D*. *clause D* = *clause a*) *Cs*) = *remove1-mset* (*clause a*) (*raw-clss-l Cs*)
⟨*proof*⟩

interpretation *raw-clss*
clause
raw-clss-l
λ*L C*. *L* ∈ *set C op* #
⟨*proof*⟩

lemma *ex-mset-unwatched-watched*:
∃ *a*. *mset* (*unwatched a*) + *mset* (*watched a*) = *E*
⟨*proof*⟩

abbreviation *conc-learned-clss* **where**
conc-learned-clss ≡ λ*S*. *mset* (*map clause* (*raw-learned-clss S*))

interpretation *twl*: *abs-state*$_W$*-ops*
clause
raw-clss-l
λ*L C*. *L* ∈ *set C op* #

mset

raw-clause λ*C*. *TWL-Clause* [] *C*
trail λ*S*. *hd* (*raw-trail S*)
(λ*S*. *raw-init-clss S* @ *raw-learned-clss S*) *backtrack-lvl raw-conflicting*
conc-learned-clss
rewrites
twl.mmset-of-mlit = *mmset-of-mlit*
⟨*proof*⟩

declare *CDCL-Two-Watched-Literals.twl.mset-ccls-ccls-of-cls*[*simp del*]

definition
candidates-propagate :: *'v twl-state* ⇒ (*'v literal* × *'v twl-clause*) *set*
where
candidates-propagate S =
{(*L*, *C*) | *L C*.
C ∈ *set* (*raw-clauses S*) ∧
set (*watched C*) − (*uminus* ‘ *lits-of-l* (*trail S*)) = {*L*} ∧
undefined-lit (*raw-trail S*) *L*}

definition *candidates-conflict* :: *'v twl-state* ⇒ *'v twl-clause set* **where**
candidates-conflict S =
{*C*. *C* ∈ *set* (*raw-clauses S*) ∧
set (*watched C*) ⊆ *uminus* ‘ *lits-of-l* (*raw-trail S*)}

primrec (*nonexhaustive*) *index* :: *'a list* ⇒*'a* ⇒ *nat* **where**
index (*a* # *l*) *c* = (*if a* = *c then 0 else 1*+*index l c*)

lemma *index-nth*:
a ∈ *set l* ⟹ *l* ! (*index l a*) = *a*
⟨*proof*⟩

Invariants

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 twl-clause* ⇒ *bool* **where**
struct-wf-twl-cls (*TWL-Clause W UW*) ⟷
distinct W ∧ *length W* ≤ *2* ∧ (*length W* < *2* ⟶ *set UW* ⊆ *set W*)

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* ⇒
'*v twl-clause* ⇒ *bool*
where
watched-only-lazy-updates M (*TWL-Clause W UW*) ⟷
(∀ *L'*∈ *set W*. ∀ *L*∈ *set UW*.
−*L'* ∈ *lits-of-l M* ⟶ −*L* ∈ *lits-of-l M* ⟶ *L* ∉ *set W* ⟶
index (*map lit-of M*) (−*L'*) ≤ *index* (*map lit-of M*) (−*L*))

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.

primrec *watched-wf-twl-cls* :: ('*a*, '*b*) *ann-lits* ⇒ '*a twl-clause* ⇒
bool **where**
watched-wf-twl-cls M (*TWL-Clause W UW*) ⟷
(∀ *L* ∈ *set W*. −*L* ∈ *lits-of-l M* ⟶ (∀ *L'* ∈ *set UW*. *L'* ∉ *set W* ⟶ −*L'* ∈ *lits-of-l M*))

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-twl-cls* :: ('*v*, '*mark*) *ann-lits* ⇒ '*v twl-clause* ⇒ *bool* **where**
wf-twl-cls M (*TWL-Clause W UW*) ⟷
struct-wf-twl-cls (*TWL-Clause W UW*) ∧ *watched-wf-twl-cls M* (*TWL-Clause W UW*) ∧
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*) ⟷ *wf-twl-cls M'* (*TWL-Clause W UW*)
⟨*proof*⟩

lemma *wf-twl-cls-wf-twl-cls-tl*:
assumes *wf*: *wf-twl-cls M C* **and** *n-d*: *no-dup M*
shows *wf-twl-cls* (*tl M*) *C*
⟨*proof*⟩

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*⟩

definition *wf-twl-state* :: '*v twl-state* ⇒ *bool* **where**
wf-twl-state S ⟷
(∀ *C* ∈ *set* (*raw-clauses S*). *wf-twl-cls* (*raw-trail S*) *C*) ∧ *no-dup* (*raw-trail S*)

lemma *wf-candidates-propagate-sound*:
assumes *wf*: *wf-twl-state S* **and**
cand: (*L*, *C*) ∈ *candidates-propagate S*
shows *raw-trail S* ⊨*as CNot* (*mset* (*removeAll L* (*raw-clause C*))) ∧ *undefined-lit* (*raw-trail S*) *L*
(**is** *?Not* ∧ *?undef*)
⟨*proof*⟩

lemma *wf-candidates-propagate-complete*:
assumes *wf*: *wf-twl-state S* **and**
c-mem: *C* ∈ *set* (*raw-clauses S*) **and**
l-mem: *L* ∈ *set* (*raw-clause C*) **and**
unsat: *trail S* ⊨*as CNot* (*mset-set* (*set* (*raw-clause C*) − {*L*})) **and**
undef: *undefined-lit* (*raw-trail S*) *L*

shows $(L,\ C) \in$ *candidates-propagate* S
⟨*proof*⟩

lemma *wf-candidates-conflict-sound*:
assumes *wf*: *wf-twl-state* S **and**
cand: $C \in$ *candidates-conflict* S
shows *trail* $S \models as$ *CNot* (*clause* C) $\wedge$ $C \in$ *set* (*raw-clauses* S)
⟨*proof*⟩

lemma *wf-candidates-conflict-complete*:
assumes *wf*: *wf-twl-state* S **and**
c-mem: $C \in$ *set* (*raw-clauses* S) **and**
unsat: *trail* $S \models as$ *CNot* (*clause* C)
shows $C \in$ *candidates-conflict* S
⟨*proof*⟩

typedef *'v wf-twl* = {*S*::*'v twl-state*. *wf-twl-state* S}
morphisms *rough-state-of-twl twl-of-rough-state*
⟨*proof*⟩

lemma [*code abstype*]:
twl-of-rough-state (*rough-state-of-twl* S) = S
⟨*proof*⟩

lemma *wf-twl-state-rough-state-of-twl*[*simp*]: *wf-twl-state* (*rough-state-of-twl* S)
⟨*proof*⟩

abbreviation *candidates-conflict-twl* :: *'v wf-twl* ⇒ *'v twl-clause set* **where**
candidates-conflict-twl $S \equiv$ *candidates-conflict* (*rough-state-of-twl* S)

abbreviation *candidates-propagate-twl* :: *'v wf-twl* ⇒ (*'v literal* × *'v twl-clause*) *set* **where**
candidates-propagate-twl $S \equiv$ *candidates-propagate* (*rough-state-of-twl* S)

abbreviation *raw-trail-twl* :: *'a wf-twl* ⇒ (*'a*, *'a twl-clause*) *ann-lits* **where**
raw-trail-twl $S \equiv$ *raw-trail* (*rough-state-of-twl* S)

abbreviation *trail-twl* :: *'a wf-twl* ⇒ (*'a*, *'a literal multiset*) *ann-lits* **where**
trail-twl $S \equiv$ *trail* (*rough-state-of-twl* S)

abbreviation *raw-clauses-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-clauses-twl $S \equiv$ *raw-clauses* (*rough-state-of-twl* S)

abbreviation *raw-init-clss-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-init-clss-twl $S \equiv$ *raw-init-clss* (*rough-state-of-twl* S)

abbreviation *raw-learned-clss-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-learned-clss-twl $S \equiv$ *raw-learned-clss* (*rough-state-of-twl* S)

abbreviation *conc-learned-clss-twl* :: *'a wf-twl* ⇒ *'a clauses* **where**
conc-learned-clss-twl $S \equiv$ *conc-learned-clss* (*rough-state-of-twl* S)

abbreviation *backtrack-lvl-twl* **where**
backtrack-lvl-twl $S \equiv$ *backtrack-lvl* (*rough-state-of-twl* S)

abbreviation *raw-conflicting-twl* **where**
raw-conflicting-twl $S \equiv$ *raw-conflicting* (*rough-state-of-twl* S)

lemma *wf-candidates-twl-conflict-complete*:
assumes
c-mem: $C \in$ *set* (*raw-clauses-twl* S) **and**
unsat: *trail-twl* $S \models as$ *CNot* (*clause* C)
shows $C \in$ *candidates-conflict-twl* S
⟨*proof*⟩

abbreviation *update-backtrack-lvl* **where**
update-backtrack-lvl k $S \equiv$
TWL-State (*raw-trail* S) (*raw-init-clss* S) (*raw-learned-clss* S) k (*raw-conflicting* S)

abbreviation *update-conflicting* **where**
update-conflicting C $S \equiv$
TWL-State (*raw-trail* S) (*raw-init-clss* S) (*raw-learned-clss* S) (*backtrack-lvl* S) C

Abstract 2-WL

definition *tl-trail* **where**
tl-trail $S =$
TWL-State (*tl* (*raw-trail* S)) (*raw-init-clss* S) (*raw-learned-clss* S) (*backtrack-lvl* S)
(*raw-conflicting* S)

locale *abstract-twl* =
fixes
watch :: $'v$ *twl-state* ⇒ $'v$ *literal list* ⇒ $'v$ *twl-clause* **and**
rewatch :: $'v$ *literal* ⇒ $'v$ *twl-state* ⇒
$'v$ *twl-clause* ⇒ $'v$ *twl-clause* **and**
restart-learned :: $'v$ *twl-state* ⇒ $'v$ *twl-clause list*
assumes
clause-watch: *no-dup* (*raw-trail* S) ⟹ *clause* (*watch* S C) = *mset* C **and**
wf-watch: *no-dup* (*raw-trail* S) ⟹ *wf-twl-cls* (*raw-trail* S) (*watch* S C) **and**
clause-rewatch: *clause* (*rewatch* L' S C') = *clause* C' **and**
wf-rewatch:
no-dup (*raw-trail* S) ⟹ *undefined-lit* (*raw-trail* S) (*lit-of* L) ⟹
wf-twl-cls (*raw-trail* S) C' ⟹
wf-twl-cls (L # *raw-trail* S) (*rewatch* (*lit-of* L) S C')
and
restart-learned: *mset* (*restart-learned* S) ⊆# *mset* (*raw-learned-clss* S) — We need *mset* and not *set* to take care of duplicates.
begin

definition
cons-trail :: ($'v$, $'v$ *twl-clause*) *ann-lit* ⇒ $'v$ *twl-state* ⇒ $'v$ *twl-state*
where
cons-trail L $S =$
TWL-State (L # *raw-trail* S) (*map* (*rewatch* (*lit-of* L) S) (*raw-init-clss* S))
(*map* (*rewatch* (*lit-of* L) S) (*raw-learned-clss* S)) (*backtrack-lvl* S) (*raw-conflicting* S)

definition
add-init-cls :: $'v$ *literal list* ⇒ $'v$ *twl-state* ⇒ $'v$ *twl-state*
where
add-init-cls C $S =$
TWL-State (*raw-trail* S) (*watch* S C # *raw-init-clss* S) (*raw-learned-clss* S) (*backtrack-lvl* S)
(*raw-conflicting* S)

definition
add-learned-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
add-learned-cls C S =
TWL-State (*raw-trail S*) (*raw-init-clss S*) (*watch S C* # *raw-learned-clss S*) (*backtrack-lvl S*)
(*raw-conflicting S*)

definition
remove-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
remove-cls C S =
TWL-State (*raw-trail S*)
(*removeAll-cond* (λ*D*. *clause D* = *mset C*) (*raw-init-clss S*))
(*removeAll-cond* (λ*D*. *clause D* = *mset C*) (*raw-learned-clss S*))
(*backtrack-lvl S*)
(*raw-conflicting S*)

definition *init-state* :: *'v literal list list* ⇒ *'v twl-state* **where**
init-state N = *fold add-init-cls N* (*TWL-State* [] [] [] *0 None*)

lemma *unchanged-fold-add-init-cls*:
raw-trail (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *M*
raw-learned-clss (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *U*
backtrack-lvl (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *k*
raw-conflicting (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *C*
⟨*proof*⟩

lemma *unchanged-init-state*[*simp*]:
raw-trail (*init-state N*) = []
raw-learned-clss (*init-state N*) = []
backtrack-lvl (*init-state N*) = *0*
raw-conflicting (*init-state N*) = *None*
⟨*proof*⟩

lemma *conc-init-clss*[*simp*]:
twl.conc-init-clss (*TWL-State M N U k C*) = *raw-clss-l N*
⟨*proof*⟩

lemma *clauses-init-fold-add-init*:
no-dup M ⟹
twl.conc-init-clss (*fold add-init-cls Cs* (*TWL-State M N U k C*)) =
clauses-of-l Cs + *raw-clss-l N*
⟨*proof*⟩

lemma *init-clss-init-state*[*simp*]: *twl.conc-init-clss* (*init-state N*) = *clauses-of-l N*
⟨*proof*⟩

definition *restart'* **where**
restart' S = *TWL-State* [] (*raw-init-clss S*) (*restart-learned S*) *0 None*

end

Instanciation of the previous locale

definition *watch-nat* :: *'v twl-state* ⇒ *'v literal list* ⇒ *'v twl-clause* **where**
watch-nat S C =

(*let*
C′ = remdups C;
neg-not-assigned = filter (λ*L*. −*L* ∉ *lits-of-l* (*raw-trail S*)) *C′*;
neg-assigned-sorted-by-trail = filter (λ*L*. *L* ∈ *set C*) (*map* (λ*L*. −*lit-of L*) (*raw-trail S*));
W = take 2 (*neg-not-assigned* @ *neg-assigned-sorted-by-trail*);
UW = foldr remove1 W C
in TWL-Clause W UW)

lemma *list-cases2*:
fixes *l* :: *'a list*
assumes
l = [] ⟹ *P* **and**
⋀*x*. *l* = [*x*] ⟹ *P* **and**
⋀*x y xs*. *l* = *x* # *y* # *xs* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *filter-in-list-prop-verifiedD*:
assumes [*L*←*P* . *Q L*] = *l*
shows ∀ *x* ∈ *set l*. *x* ∈ *set P* ∧ *Q x*
⟨*proof*⟩

lemma *no-dup-filter-diff*:
assumes *n-d*: *no-dup M* **and** *H*: [*L*←*map* (λ*L*. − *lit-of L*) *M*. *L* ∈ *set C*] = *l*
shows *distinct l*
⟨*proof*⟩

lemma *watch-nat-lists-disjointD*:
assumes
l: [*L*←*remdups C*. − *L* ∉ *lits-of-l* (*raw-trail S*)] = *l* **and**
l′: [*L*←*map* (λ*L*. − *lit-of L*) (*raw-trail S*) . *L* ∈ *set C*] = *l′*
shows ∀ *x* ∈ *set l*. ∀ *y* ∈ *set l′*. *x* ≠ *y*
⟨*proof*⟩

lemma *watch-nat-list-cases-witness*[*consumes 2*, *case-names Nil-Nil Nil-single Nil-other single-Nil single-other other*]:
fixes
C :: *'v literal list* **and**
S :: *'v twl-state*
defines
xs ≡ [*L*←*remdups C*. − *L* ∉ *lits-of-l* (*raw-trail S*)] **and**
ys ≡ [*L*←*map* (λ*L*. − *lit-of L*) (*raw-trail S*) . *L* ∈ *set C*]
assumes
n-d: *no-dup* (*raw-trail S*) **and**
Nil-Nil: *xs* = [] ⟹ *ys* = [] ⟹ *P* **and**
Nil-single:
⋀*a*. *xs* = [] ⟹ *ys* = [*a*] ⟹ *a* ∈ *set C* ⟹ *P* **and**
Nil-other: ⋀*a b ys′*. *xs* = [] ⟹ *ys* = *a* # *b* # *ys′* ⟹ *a* ≠ *b* ⟹ *P* **and**
single-Nil: ⋀*a*. *xs* = [*a*] ⟹ *ys* = [] ⟹ *P* **and**
single-other: ⋀*a b ys′*. *xs* = [*a*] ⟹ *ys* = *b* # *ys′* ⟹ *a* ≠ *b* ⟹ *P* **and**
other: ⋀*a b xs′*. *xs* = *a* # *b* # *xs′* ⟹ *a* ≠ *b* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *watch-nat-list-cases* [*consumes 1*, *case-names Nil-Nil Nil-single Nil-other single-Nil single-other other*]:

fixes
$C :: 'v\ literal\ list$ **and**
$S :: 'v\ twl\text{-}state$
defines
$xs \equiv [L{\leftarrow}remdups\ C\ .\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L{\leftarrow}map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
assumes
n-d: $no\text{-}dup\ (raw\text{-}trail\ S)$ **and**
Nil-Nil: $xs = [] \Longrightarrow ys = [] \Longrightarrow P$ **and**
Nil-single:
$\bigwedge a.\ xs = [] \Longrightarrow ys = [a] \Longrightarrow\ a \in set\ C \Longrightarrow P$ **and**
Nil-other: $\bigwedge a\ b\ ys'.\ xs = [] \Longrightarrow ys = a \#\ b \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
single-Nil: $\bigwedge a.\ xs = [a] \Longrightarrow ys = [] \Longrightarrow P$ **and**
single-other: $\bigwedge a\ b\ ys'.\ xs = [a] \Longrightarrow ys = b \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
other: $\bigwedge a\ b\ xs'.\ xs = a \#\ b \#\ xs' \Longrightarrow a \neq b \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *watch-nat-lists-set-union-witness*:
fixes
$C ::\ 'v\ literal\ list$ **and**
$S :: 'v\ twl\text{-}state$
defines
$xs \equiv [L{\leftarrow}remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L{\leftarrow}map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
assumes *n-d*: $no\text{-}dup\ (raw\text{-}trail\ S)$
shows $set\ C = set\ xs \cup set\ ys$
⟨*proof*⟩

lemma *mset-intersection-inclusion*: $A + (B - A) = B \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *clause-watch-nat*:
assumes $no\text{-}dup\ (raw\text{-}trail\ S)$
shows $clause\ (watch\text{-}nat\ S\ C) = mset\ C$
⟨*proof*⟩

lemma *index-uminus-index-map-uminus*:
$-a \in set\ L \Longrightarrow index\ L\ (-a) = index\ (map\ uminus\ L)\ (a::'a\ literal)$
⟨*proof*⟩

lemma *index-filter*:
$a \in set\ L \Longrightarrow b \in set\ L \Longrightarrow P\ a \Longrightarrow P\ b \Longrightarrow$
$index\ L\ a \leq index\ L\ b \longleftrightarrow index\ (filter\ P\ L)\ a \leq index\ (filter\ P\ L)\ b$
⟨*proof*⟩

lemma *foldr-remove1-W-Nil*[*simp*]: $foldr\ remove1\ W\ [] = []$
⟨*proof*⟩

lemma *image-lit-of-mmset-of-mlit*[*simp*]:
$lit\text{-}of\ `\ mmset\text{-}of\text{-}mlit\ `\ A = lit\text{-}of\ `\ A$
⟨*proof*⟩

lemma *distinct-filter-eq*:
assumes $distinct\ xs$
shows $[L{\leftarrow}\ xs.\ L = a] = (if\ a \in set\ xs\ then\ [a]\ else\ [])$

⟨*proof*⟩

lemma *no-dup-distinct-map-uminus-lit-of*:
no-dup xs ⟹ *distinct* (*map* (λ*L*. − *lit-of L*) *xs*)
⟨*proof*⟩

lemma *wf-watch-witness*:
fixes *C* :: *'v literal list* **and**
S :: *'v twl-state*
defines
ass: *neg-not-assigned* ≡ *filter* (λ*L*. −*L* ∉ *lits-of-l* (*raw-trail S*)) (*remdups C*) **and**
tr: *neg-assigned-sorted-by-trail* ≡ *filter* (λ*L*. *L* ∈ *set C*) (*map* (λ*L*. −*lit-of L*) (*raw-trail S*))
defines
W: *W* ≡ *take 2* (*neg-not-assigned* @ *neg-assigned-sorted-by-trail*)
assumes
n-d[*simp*]: *no-dup* (*raw-trail S*)
shows *wf-twl-cls* (*raw-trail S*) (*TWL-Clause W* (*foldr remove1 W C*))
⟨*proof*⟩

lemma *wf-watch-nat*: *no-dup* (*raw-trail S*) ⟹ *wf-twl-cls* (*raw-trail S*) (*watch-nat S C*)
⟨*proof*⟩

definition
rewatch-nat ::
'v literal ⇒ *'v twl-state* ⇒ *'v twl-clause* ⇒ *'v twl-clause*
where
rewatch-nat L S C =
(*if* − *L* ∈ *set* (*watched C*) *then*
case filter (λ*L'*. *L'* ∉ *set* (*watched C*) ∧ − *L'* ∉ *insert L* (*lits-of-l* (*trail S*)))
(*unwatched C*) *of*
[] ⇒ *C*
| *L'* # - ⇒
TWL-Clause (*L'* # *remove1* (−*L*) (*watched C*)) (−*L* # *remove1 L'* (*unwatched C*))
else
C)

lemma *clause-rewatch-nat*:
fixes *UW* :: *'v literal list* **and**
S :: *'v twl-state* **and**
L :: *'v literal* **and** *C* :: *'v twl-clause*
shows *clause* (*rewatch-nat L S C*) = *clause C*
⟨*proof*⟩

lemma *filter-sorted-list-of-multiset-Nil*:
[*x* ← *sorted-list-of-multiset M*. *p x*] = [] ⟷ (∀ *x* ∈# *M*. ¬ *p x*)
⟨*proof*⟩

lemma *filter-sorted-list-of-multiset-ConsD*:
[*x* ← *sorted-list-of-multiset M*. *p x*] = *x* # *xs* ⟹ *p x*
⟨*proof*⟩

lemma *mset-minus-single-eq-mempty*:
a − {#*b*#} = {#} ⟷ *a* = {#*b*#} ∨ *a* = {#}
⟨*proof*⟩

lemma *size-mset-le-2-cases*:

assumes *size W* ≤ *2*
shows *W* = {#} ∨ (∃ *a*. *W* = {#*a*#}) ∨ (∃ *a b*. *W* = {#*a*,*b*#})
⟨*proof*⟩

lemma *filter-sorted-list-of-multiset-eqD*:
assumes [*x* ← *sorted-list-of-multiset A*. *p x*] = *x* # *xs* (**is** *?comp* = -)
shows *x* ∈# *A*
⟨*proof*⟩

lemma *clause-rewatch-witness'*:
assumes
wf: *wf-twl-cls* (*raw-trail S*) *C* **and**
undef: *undefined-lit* (*raw-trail S*) (*lit-of L*)
shows *wf-twl-cls* (*L* # *raw-trail S*) (*rewatch-nat* (*lit-of L*) *S C*)
⟨*proof*⟩

interpretation *twl*: *abstract-twl watch-nat rewatch-nat raw-learned-clss*
⟨*proof*⟩

interpretation *twl2*: *abstract-twl watch-nat rewatch-nat* λ-. []
⟨*proof*⟩

end

7.3.2 Two Watched-Literals with invariant

theory *CDCL-Two-Watched-Literals-Invariant*
imports *CDCL-Two-Watched-Literals DPLL-CDCL-W-Implementation*
begin

Interpretation for *conflict-driven-clause-learning*$_W$*.cdcl*$_W$

We define here the 2-WL with the invariant of well-foundedness and show the role of the candidates by defining an equivalent CDCL procedure using the candidates given by the datastructure.

context *abstract-twl*
begin

Direct Interpretation **lemma** *mset-map-removeAll-cond*:
mset (*map clause*
(*removeAll-cond* (λ*D*. *clause D* = *clause C*) *N*))
= *mset* (*removeAll* (*clause C*) (*map clause N*))
⟨*proof*⟩

lemma *raw-clss-l-raw-init-clss-conc-init-clss*[*simp*]:
raw-clss-l (*raw-init-clss S*) = *twl.conc-init-clss S*
⟨*proof*⟩
lemma *mset-raw-init-clss-init-state*:
raw-clss-l (*raw-init-clss* (*init-state* (*map raw-clause N*))) = *raw-clss-l N*
⟨*proof*⟩

fun *reduce-trail-to* **where**
reduce-trail-to M1 S =

(*case S of*
(*TWL-State M N U k C*) ⇒ *TWL-State* (*drop* (*length M* − *length M1*) *M*) *N U k C*)

abbreviation *resolve-conflicting* **where**
resolve-conflicting L D S ≡
update-conflicting
(*Some* (*union-mset-list* (*remove1* (−*L*) (*the* (*raw-conflicting S*))) (*remove1 L* (*raw-clause D*))))
S

interpretation *rough-cdcl*: *abs-state*$_W$*-ops*
clause
raw-clss-l
λ*L C*. *L* ∈ *set C op* #

mset

raw-clause λ*C*. *TWL-Clause* [] *C*
trail λ*S*. *hd* (*raw-trail S*)
(λ*S*. *raw-init-clss S* @ *raw-learned-clss S*) *backtrack-lvl raw-conflicting*
conc-learned-clss

cons-trail tl-trail λ*S*. *update-conflicting None* (*add-learned-cls* (*the* (*raw-conflicting S*)) *S*)
λ*C*. *remove-cls* (*raw-clause C*)
update-backtrack-lvl
λ*C*. *update-conflicting* (*Some C*) *reduce-trail-to resolve-conflicting*
λ*N*. *init-state* (*map raw-clause N*) *restart*′
rewrites
rough-cdcl.mmset-of-mlit = *mmset-of-mlit*
⟨*proof*⟩

interpretation *rough-cdcl*: *abs-state*$_W$
clause
raw-clss-l
λ*L C*. *L* ∈ *set C op* #

mset

raw-clause λ*C*. *TWL-Clause* [] *C*
trail λ*S*. *hd* (*raw-trail S*)
(λ*S*. *raw-init-clss S* @ *raw-learned-clss S*) *backtrack-lvl raw-conflicting*
conc-learned-clss

cons-trail tl-trail λ*S*. *update-conflicting None* (*add-learned-cls* (*the* (*raw-conflicting S*)) *S*)
λ*C*. *remove-cls* (*raw-clause C*)
update-backtrack-lvl
λ*C*. *update-conflicting* (*Some C*) *reduce-trail-to resolve-conflicting*
λ*N*. *init-state* (*map raw-clause N*) *restart*′
⟨*proof*⟩

interpretation *rough-cdcl*: *abs-conflict-driven-clause-learning*$_W$
clause
raw-clss-l
λ*L C*. *L* ∈ *set C op* #

mset

raw-clause λC. *TWL-Clause* [] C
trail λS. *hd* (*raw-trail* S)
(λS. *raw-init-clss* S @ *raw-learned-clss* S) *backtrack-lvl raw-conflicting*
conc-learned-clss

cons-trail tl-trail λS. *update-conflicting None* (*add-learned-cls* (*the* (*raw-conflicting* S)) S)
λC. *remove-cls* (*raw-clause* C)
update-backtrack-lvl
λC. *update-conflicting* (*Some* C) *reduce-trail-to resolve-conflicting*
λN. *init-state* (*map raw-clause* N) *restart'*
⟨*proof*⟩

declare *local.rough-cdcl.mset-ccls-ccls-of-cls*[*simp del*]

Opaque Type with Invariant **declare** *rough-cdcl.state-simp*[*simp del*]

definition *cons-trail-twl* :: ('*v*, '*v twl-clause*) *ann-lit* ⇒ '*v wf-twl* ⇒ '*v wf-twl*
where
cons-trail-twl L S ≡ *twl-of-rough-state* (*cons-trail L* (*rough-state-of-twl S*))

lemma *wf-twl-state-cons-trail*:
assumes
undef: *undefined-lit* (*raw-trail S*) (*lit-of L*) **and**
wf: *wf-twl-state S*
shows *wf-twl-state* (*cons-trail L S*)
⟨*proof*⟩

lemma *rough-state-of-twl-cons-trail*:
undefined-lit (*raw-trail-twl S*) (*lit-of L*) ⟹
rough-state-of-twl (*cons-trail-twl L S*) = *cons-trail L* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *add-init-cls-twl* **where**
add-init-cls-twl C S ≡ *twl-of-rough-state* (*add-init-cls C* (*rough-state-of-twl S*))

lemma *wf-twl-add-init-cls*: *wf-twl-state S* ⟹ *wf-twl-state* (*add-init-cls L S*)
⟨*proof*⟩

lemma *rough-state-of-twl-add-init-cls*:
rough-state-of-twl (*add-init-cls-twl L S*) = *add-init-cls L* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *add-learned-cls-twl* **where**
add-learned-cls-twl C S ≡ *twl-of-rough-state* (*add-learned-cls C* (*rough-state-of-twl S*))

lemma *wf-twl-add-learned-cls*: *wf-twl-state S* ⟹ *wf-twl-state* (*add-learned-cls L S*)
⟨*proof*⟩

lemma *rough-state-of-twl-add-learned-cls*:
rough-state-of-twl (*add-learned-cls-twl L S*) = *add-learned-cls L* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *remove-cls-twl* **where**
remove-cls-twl C S ≡ *twl-of-rough-state* (*remove-cls C* (*rough-state-of-twl S*))

lemma *set-removeAll-condD*: *x* ∈ *set* (*removeAll-cond f xs*) ⟹ *x* ∈ *set xs*

⟨*proof*⟩

lemma *wf-twl-remove-cls*: *wf-twl-state S* ⟹ *wf-twl-state* (*remove-cls L S*)
⟨*proof*⟩

lemma *rough-state-of-twl-remove-cls*:
rough-state-of-twl (*remove-cls-twl L S*) = *remove-cls L* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *init-state-twl* **where**
init-state-twl N ≡ *twl-of-rough-state* (*init-state N*)

lemma *wf-twl-state-wf-twl-state-fold-add-init-cls*:
assumes *wf-twl-state S*
shows *wf-twl-state* (*fold add-init-cls N S*)
⟨*proof*⟩

lemma *wf-twl-state-epsilon-state*[*simp*]:
wf-twl-state (*TWL-State* [] [] [] *0 None*)
⟨*proof*⟩

lemma *wf-twl-init-state*: *wf-twl-state* (*init-state N*)
⟨*proof*⟩

lemma *rough-state-of-twl-init-state*:
rough-state-of-twl (*init-state-twl N*) = *init-state N*
⟨*proof*⟩

abbreviation *tl-trail-twl* **where**
tl-trail-twl S ≡ *twl-of-rough-state* (*tl-trail* (*rough-state-of-twl S*))

lemma *wf-twl-state-tl-trail*: *wf-twl-state S* ⟹ *wf-twl-state* (*tl-trail S*)
⟨*proof*⟩

lemma *rough-state-of-twl-tl-trail*:
rough-state-of-twl (*tl-trail-twl S*) = *tl-trail* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *update-backtrack-lvl-twl* **where**
update-backtrack-lvl-twl k S ≡ *twl-of-rough-state* (*update-backtrack-lvl k* (*rough-state-of-twl S*))

lemma *wf-twl-state-update-backtrack-lvl*:
wf-twl-state S ⟹ *wf-twl-state* (*update-backtrack-lvl k S*)
⟨*proof*⟩

lemma *rough-state-of-twl-update-backtrack-lvl*:
rough-state-of-twl (*update-backtrack-lvl-twl k S*) = *update-backtrack-lvl k*
(*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *update-conflicting-twl* **where**
update-conflicting-twl k S ≡ *twl-of-rough-state* (*update-conflicting k* (*rough-state-of-twl S*))

lemma *wf-twl-state-update-conflicting*:
wf-twl-state S ⟹ *wf-twl-state* (*update-conflicting k S*)
⟨*proof*⟩

lemma *rough-state-of-twl-update-add-learned-cls*:
  rough-state-of-twl (*update-conflicting-twl None* (*add-learned-cls-twl C S*)) =
    update-conflicting None (*add-learned-cls C* (*rough-state-of-twl S*))
    (**is** *rough-state-of-twl ?upd = update-conflicting None ?le*)
  ⟨*proof*⟩

abbreviation *reduce-trail-to-twl* **where**
reduce-trail-to-twl M1 S ≡ *twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl S*))

abbreviation *resolve-conflicting-twl* **where**
resolve-conflicting-twl L D S ≡
  twl-of-rough-state (*resolve-conflicting L D* (*rough-state-of-twl S*))

lemma *rough-state-of-twl-update-conflicting*:
  rough-state-of-twl (*update-conflicting-twl k S*) = *update-conflicting k*
    (*rough-state-of-twl S*)
  ⟨*proof*⟩

abbreviation *raw-clauses-twl* **where**
raw-clauses-twl S ≡ *raw-clauses* (*rough-state-of-twl S*)

abbreviation *restart-twl* **where**
restart-twl S ≡ *twl-of-rough-state* (*restart′* (*rough-state-of-twl S*))
lemma *mset-union-mset-setD*:
  mset A ⊆# *mset B* ⟹ *set A* ⊆ *set B*
  ⟨*proof*⟩

lemma *wf-wf-restart′*: *wf-twl-state S* ⟹ *wf-twl-state* (*restart′ S*)
  ⟨*proof*⟩

lemma *rough-state-of-twl-restart-twl*:
  rough-state-of-twl (*restart-twl S*) = *restart′* (*rough-state-of-twl S*)
  ⟨*proof*⟩

lemma *undefined-lit-trail-twl-raw-trail*[*iff*]:
  undefined-lit (*trail-twl S*) *L* ⟷ *undefined-lit* (*raw-trail-twl S*) *L*
  ⟨*proof*⟩

lemma *wf-twl-reduce-trail-to*:
  assumes *trail S = M2 @ M1* **and** *wf*: *wf-twl-state S*
  shows *wf-twl-state* (*reduce-trail-to M1 S*)
⟨*proof*⟩

lemma *trail-twl-twl-rough-state-reduce-trail-to*:
  assumes *trail-twl st = M2 @ M1*
  shows *trail-twl* (*twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl st*))) = *M1*
⟨*proof*⟩

lemma *twl-of-rough-state-reduce-trail-to*:
  assumes *trail-twl st = M2 @ M1* **and**
    S: *rough-cdcl.state* (*rough-state-of-twl st*) = (*M*, *S*)
  shows
    rough-cdcl.state
      (*rough-state-of-twl* (*twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl st*)))) =

(*M1*, *S*) (**is** *?st*) **and**
raw-init-clss-twl (*twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl st*)))
= *raw-init-clss-twl st* (**is** *?A*) **and**
raw-learned-clss-twl (*twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl st*)))
= *raw-learned-clss-twl st* (**is** *?B*) **and**
backtrack-lvl-twl (*twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl st*)))
= *backtrack-lvl-twl st* (**is** *?C*) **and**
rough-cdcl.conc-conflicting (*rough-state-of-twl* (*twl-of-rough-state*
(*reduce-trail-to M1* (*rough-state-of-twl st*))))
= *rough-cdcl.conc-conflicting* (*rough-state-of-twl st*) (**is** *?D*)
⟨*proof*⟩

lemma *add-learned-cls-rough-state-of-twl-simp*:
assumes *raw-conflicting-twl st* = *Some z*
shows
trail (*add-learned-cls z* (*rough-state-of-twl st*)) = *trail-twl st*
rough-cdcl.conc-init-clss (*add-learned-cls z* (*rough-state-of-twl st*)) =
rough-cdcl.conc-init-clss (*rough-state-of-twl st*)
conc-learned-clss (*local.add-learned-cls z* (*rough-state-of-twl st*)) =
{#*mset z*#} + *conc-learned-clss* (*rough-state-of-twl st*)
backtrack-lvl (*add-learned-cls z* (*rough-state-of-twl st*)) = *backtrack-lvl-twl st*
⟨*proof*⟩

sublocale *wf-twl*: *abs-state*$_W$*-ops*
clause
raw-clss-l
λ*L C*. *L* ∈ *set C op* #

mset

λ*C*. *raw-clause C* λ*C*. *TWL-Clause* [] *C*
trail-twl λ*S*. *hd* (*raw-trail-twl S*)
raw-clauses-twl
backtrack-lvl-twl
raw-conflicting-twl
conc-learned-clss-twl

cons-trail-twl
tl-trail-twl
λ*S*. *update-conflicting-twl None* (*add-learned-cls-twl* (*the* (*raw-conflicting-twl S*)) *S*)
λ*C*. *remove-cls-twl* (*raw-clause C*)
update-backtrack-lvl-twl
λ*C*. *update-conflicting-twl* (*Some C*)
reduce-trail-to-twl
resolve-conflicting-twl
λ*N*. *init-state-twl* (*map raw-clause N*)
restart-twl
⟨*proof*⟩

lemma *wf-twl-conc-init-clss-restart-twl*[*simp*]:
wf-twl.conc-init-clss (*restart-twl S*) = *wf-twl.conc-init-clss S*
⟨*proof*⟩

sublocale *wf-twl*: *abs-state*$_W$
clause
raw-clss-l

λL C. L ∈ set C op #

mset

λC. raw-clause C λC. TWL-Clause [] C
trail-twl λS. hd (raw-trail-twl S)
raw-clauses-twl
backtrack-lvl-twl
raw-conflicting-twl
conc-learned-clss-twl

cons-trail-twl
tl-trail-twl
λS. update-conflicting-twl None (add-learned-cls-twl (the (raw-conflicting-twl S)) S)
λC. remove-cls-twl (raw-clause C)
update-backtrack-lvl-twl
λC. update-conflicting-twl (Some C)
reduce-trail-to-twl
resolve-conflicting-twl
λN. init-state-twl (map raw-clause N)
restart-twl
⟨*proof*⟩

sublocale *wf-twl*: *abs-conflict-driven-clause-learning*$_W$
clause
raw-clss-l
λL C. L ∈ set C op #

mset

λC. raw-clause C λC. TWL-Clause [] C
trail-twl λS. hd (raw-trail-twl S)
raw-clauses-twl
backtrack-lvl-twl
raw-conflicting-twl
conc-learned-clss-twl

cons-trail-twl
tl-trail-twl
λS. update-conflicting-twl None (add-learned-cls-twl (the (raw-conflicting-twl S)) S)
λC. remove-cls-twl (raw-clause C)
update-backtrack-lvl-twl
λC. update-conflicting-twl (Some C)
reduce-trail-to-twl
resolve-conflicting-twl
λN. init-state-twl (map raw-clause N)
restart-twl
⟨*proof*⟩

declare *local.rough-cdcl.mset-ccls-ccls-of-cls*[*simp del*]
abbreviation *state-eq-twl* (**infix** ∼*TWL 51*) **where**
state-eq-twl S S′ ≡ *rough-cdcl.state-eq* (*rough-state-of-twl S*) (*rough-state-of-twl S′*)
notation *wf-twl.state-eq* (**infix** ∼ *51*)

To avoid ambiguities:

no-notation *state-eq-twl* (**infix** $\sim$ *51*)

Alternative Definition of CDCL using the candidates of 2-WL **inductive** *propagate-twl* :: *'v wf-twl* $\Rightarrow$ *'v wf-twl* $\Rightarrow$ *bool* **where**
propagate-twl-rule: $(L, C) \in$ *candidates-propagate-twl S* $\Longrightarrow$
$S' \sim$ *cons-trail-twl* (*Propagated L C*) *S* $\Longrightarrow$
raw-conflicting-twl S = *None* $\Longrightarrow$
propagate-twl S S'

lemma *cdcl$_W$-all-struct-inv-clause-distinct-mset*:
cdcl$_W$-mset.cdcl$_W$-all-struct-inv (*wf-twl.state S*) $\Longrightarrow$
$C \in$ *set* (*CDCL-Two-Watched-Literals.raw-clauses-twl S*) $\Longrightarrow$ *distinct* (*raw-clause C*)
⟨*proof*⟩

inductive-cases *propagate-twlE*: *propagate-twl S T*
lemma *propagate-twl-iff-propagate*:
assumes *inv*: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*wf-twl.state S*)
shows *wf-twl.propagate-abs S T* $\longleftrightarrow$ *propagate-twl S T* (**is** *?P* $\longleftrightarrow$ *?T*)
⟨*proof*⟩

no-notation *twl.state-eq-twl* (**infix** $\sim$*TWL 51*)

inductive *conflict-twl* **where**
conflict-twl-rule:
$C \in$ *candidates-conflict-twl S* $\Longrightarrow$
$S' \sim$ *update-conflicting-twl* (*Some* (*raw-clause C*)) *S* $\Longrightarrow$
raw-conflicting-twl S = *None* $\Longrightarrow$
conflict-twl S S'

inductive-cases *conflict-twlE*: *conflict-twl S T*

lemma *conflict-twl-iff-conflict*:
shows *wf-twl.conflict-abs S T* $\longleftrightarrow$ *conflict-twl S T* (**is** *?C* $\longleftrightarrow$ *?T*)
⟨*proof*⟩

We have shown that we we can use *conflict-twl* and *propagate-twl* in a CDCL calculus.

end

end