

# Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

April 20, 2016



# Contents

<b>1</b>	<b>More Standard Theorems</b>	<b>7</b>
1.1	More about Multisets	7
1.1.1	Basic Setup	7
1.1.2	Lemmas about intersections	8
1.1.3	Lemmas about size	8
1.1.4	Multiset Extension of Multiset Ordering	9
1.1.5	Remove	10
1.1.6	Replicate	10
1.1.7	Multiset and set conversion	11
1.1.8	Removing duplicates	11
1.1.9	Filter	14
1.1.10	Sums	14
1.1.11	Order	15
1.2	Transitions	15
1.2.1	More theorems about Closures	15
1.2.2	Full Transitions	16
1.2.3	Well-Foundedness and Full Transitions	17
1.2.4	More Well-Foundedness	18
1.3	Various Lemmas	19
1.4	More List	20
1.4.1	<i>upt</i>	20
1.4.2	Lexicographic Ordering	21
1.4.3	Remove	22
<b>2</b>	<b>Definition of Entailment</b>	<b>25</b>
2.1	Clausal Logic	25
2.1.1	Literals	25
2.1.2	Clauses	27
2.2	Clausal Logic	29
2.2.1	Herbrand Interpretations	29
2.3	Partial Clausal Logic	31
2.3.1	Clauses	31
2.3.2	Partial Interpretations	31
2.3.3	Subsumptions	43
2.3.4	Removing Duplicates	43
2.3.5	Set of all Simple Clauses	43
2.3.6	Experiment: Expressing the Entailments as Locales	45
2.3.7	Entailment to be extended	45

<b>3</b>	<b>Normalisation</b>	<b>47</b>
3.1	Logics	47
3.1.1	Definition and abstraction	47
3.1.2	properties of the abstraction	48
3.1.3	Subformulas and properties	50
3.1.4	Positions	52
3.2	Semantics over the syntax	53
3.3	Rewrite systems and properties	54
3.3.1	Lifting of rewrite rules	54
3.3.2	Consistency preservation	55
3.3.3	Full Lifting	56
3.4	Transformation testing	56
3.4.1	Definition and first properties	56
3.4.2	Invariant conservation	57
3.5	Rewrite Rules	59
3.5.1	Elimination of the equivalences	59
3.5.2	Eliminate Implication	61
3.5.3	Eliminate all the True and False in the formula	62
3.5.4	PushNeg	66
3.5.5	Push inside	68
3.6	The full transformations	73
3.6.1	Abstract Property characterizing that only some connective are inside the others	73
3.6.2	Conjunctive Normal Form	74
3.6.3	Disjunctive Normal Form	75
3.7	More aggressive simplifications: Removing true and false at the beginning	75
3.7.1	Transformation	75
3.7.2	More invariants	76
3.7.3	The new CNF and DNF transformation	77
3.8	Link with Multiset Version	78
3.8.1	Transformation to Multiset	78
3.8.2	Equisatisfiability of the two Version	78
<b>4</b>	<b>Resolution-based techniques</b>	<b>81</b>
4.1	Resolution	81
4.1.1	Simplification Rules	81
4.1.2	Unconstrained Resolution	82
4.1.3	Inference Rule	83
4.1.4	Lemma about the simplified state	88
4.1.5	Resolution and Invariants	89
4.2	Superposition	97
4.2.1	We can now define the rules of the calculus	102
4.3	Partial Clausal Logic	104
4.3.1	Decided Literals	104
4.3.2	Backtracking	108
4.3.3	Decomposition with respect to the First Decided Literals	108
4.3.4	Negation of Clauses	112
4.3.5	Other	114
4.3.6	Extending Entailments to multisets	115

<b>5</b>	<b>NOT's CDCL and DPLL</b>	<b>117</b>
5.1	Measure . . . . .	117
5.2	NOT's CDCL . . . . .	119
5.2.1	Auxiliary Lemmas and Measure . . . . .	119
5.2.2	Initial definitions . . . . .	119
5.2.3	DPLL with backjumping . . . . .	124
5.2.4	CDCL . . . . .	130
5.2.5	CDCL with restarts . . . . .	141
5.2.6	Merging backjump and learning . . . . .	146
5.2.7	Instantiations . . . . .	150
5.3	DPLL as an instance of NOT . . . . .	156
5.3.1	DPLL with simple backtrack . . . . .	156
5.3.2	Adding restarts . . . . .	158
5.4	Weidenbach's DPLL . . . . .	159
5.4.1	Rules . . . . .	159
5.4.2	Invariants . . . . .	159
5.4.3	Termination . . . . .	162
5.4.4	Final States . . . . .	162
5.4.5	Link with NOT's DPLL . . . . .	163
<b>6</b>	<b>Weidenbach's CDCL</b>	<b>167</b>
6.1	Weidenbach's CDCL with Multisets . . . . .	167
6.1.1	The State . . . . .	167
6.1.2	CDCL Rules . . . . .	174
6.1.3	Structural Invariants . . . . .	180
6.1.4	CDCL Strong Completeness . . . . .	190
6.1.5	Higher level strategy . . . . .	191
6.1.6	Termination . . . . .	204
6.2	Merging backjump rules . . . . .	211
6.2.1	Inclusion of the states . . . . .	211
6.2.2	More lemmas conflict-propagate and backjumping . . . . .	212
6.2.3	CDCL with Merging . . . . .	214
6.2.4	CDCL with Merge and Strategy . . . . .	216
6.3	Link between Weidenbach's and NOT's CDCL . . . . .	223
6.3.1	Inclusion of the states . . . . .	223
6.3.2	Additional Lemmas between NOT and W states . . . . .	226
6.3.3	Inclusion of Weidenbach's CDCL in NOT's CDCL . . . . .	227
6.3.4	Correctness of <i>cdcl<sub>W</sub>-merge-stgy</i> . . . . .	227
6.3.5	Adding Restarts . . . . .	228
6.4	Incremental SAT solving . . . . .	232
<b>7</b>	<b>Implementation of DPLL and CDCL</b>	<b>239</b>
7.1	Simple List-Based Implementation of the DPLL and CDCL . . . . .	239
7.1.1	Common Rules . . . . .	239
7.1.2	CDCL specific functions . . . . .	241
7.1.3	Simple Implementation of DPLL . . . . .	242
7.1.4	List-based CDCL Implementation . . . . .	246
7.1.5	CDCL Implementation . . . . .	248
7.1.6	Abstract Clause Representation . . . . .	260
7.2	Weidenbach's CDCL with Abstract Clause Representation . . . . .	262

7.2.1	Instantiation of the Multiset Version . . . . .	262
7.2.2	Abstract Relation and Relation Theorems . . . . .	264
7.2.3	The State . . . . .	266
7.2.4	CDCL Rules . . . . .	274
7.2.5	Higher level strategy . . . . .	281
7.3	2-Watched-Literal . . . . .	282
7.3.1	Essence of 2-WL . . . . .	283
7.3.2	Two Watched-Literals with invariant . . . . .	292

# 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]:
  ( $\bigwedge x. x \in \# M \implies f x = g x \implies M = N \implies \text{image-mset } f M = \text{image-mset } g N$ )
  <proof>
```

```
lemma subset-msetE [elim!]:
  ( $[A \subset \# B; [A \subseteq \# B; \sim (B \subseteq \# A)]] \implies R$ )  $\implies R$ 
  <proof>
```

```
lemma ball-msetE [elim]:  $\forall x \in \# A. P x \implies (P x \implies Q) \implies (x \notin \# A \implies Q) \implies Q$ 
  <proof>
```

**lemma** *bex-msetI* [intro]:  $P\ x \implies x \in \#A \implies \exists x \in \#A. P\ x$   
 — Normally the best argument order:  $P\ x$  constrains the choice of  $x \in \#A$ .  
 $\langle proof \rangle$

**lemma** *rev-bex-msetI* [intro]:  $x \in \#A \implies P\ x \implies \exists x \in \#A. P\ x$   
 — The best argument order when there is only one  $x \in \#A$ .  
 $\langle proof \rangle$

### 1.1.2 Lemmas about intersections

**lemma** *mset-inter-single*:  
 $x \in \# \Sigma \implies \Sigma \ \# \cap \ \{ \#x\# \} = \{ \#x\# \}$   
 $x \notin \# \Sigma \implies \Sigma \ \# \cap \ \{ \#x\# \} = \{ \# \}$   
 $\langle proof \rangle$

### 1.1.3 Lemmas about size

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

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

**lemma** *size-Suc-Diff1*:  
 $x \in \# \Sigma \implies Suc\ (size\ (\Sigma - \{ \#x\# \})) = size\ \Sigma$   
 $\langle proof \rangle$

**lemma** *size-Diff-singleton*:  $x \in \# \Sigma \implies size\ (\Sigma - \{ \#x\# \}) = size\ \Sigma - 1$   
 $\langle proof \rangle$

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

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

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

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

**lemma** *diff-size-le-size-Diff*:  $size\ (\Sigma :: \text{multiset}) - size\ \Sigma' \leq size\ (\Sigma - \Sigma')$   
 $\langle proof \rangle$

**lemma** *size-Diff1-less*:  $x \in \# \Sigma \implies size\ (\Sigma - \{ \#x\# \}) < size\ \Sigma$   
 $\langle proof \rangle$

**lemma** *size-Diff2-less*:  $x \in \# \Sigma \implies y \in \# \Sigma \implies size\ (\Sigma - \{ \#x\# \} - \{ \#y\# \}) < size\ \Sigma$   
 $\langle proof \rangle$

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



$\langle \text{proof} \rangle$

**lemma** *size-psubset*:  $(\Sigma :: \text{multiset}) \leq_{\#} \Sigma' \implies \text{size } \Sigma < \text{size } \Sigma' \implies \Sigma <_{\#} \Sigma'$

$\langle \text{proof} \rangle$

### 1.1.4 Multiset Extension of Multiset Ordering

The  $op \#<_{\#}$  and  $op \# \subseteq_{\#}$  operators are introduced as the multiset extension of the multiset orderings of  $op \#<_{\#}$  and  $op \# \subseteq_{\#}$ .

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

**where**

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

**definition** *le-mset-mset* ::  $('a :: \text{order}) \text{multiset multiset} \Rightarrow 'a \text{multiset multiset} \Rightarrow \text{bool}$   
 $(\text{infix } \# \leq_{\#} 50)$

**where**

$M' \# \leq_{\#} M \longleftrightarrow M' \#<_{\#} M \vee M' = M$

**notation** *less-mset-mset* (**infix**  $\#<_{\#}$  50)

**notation** *le-mset-mset* (**infix**  $\# \leq_{\#}$  50)

**lemmas** *less-mset-mset<sub>DM</sub>* = *order.mult<sub>DM</sub>*[*OF order-multiset, folded less-mset-mset-def*]

**lemmas** *less-mset-mset<sub>HO</sub>* = *order.mult<sub>HO</sub>*[*OF order-multiset, folded less-mset-mset-def*]

**interpretation** *multiset-multiset-order*: *order*

*le-mset-mset* ::  $('a :: \text{linorder}) \text{multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{multiset multiset} \Rightarrow \text{bool}$

*less-mset-mset* ::  $('a :: \text{linorder}) \text{multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{multiset multiset} \Rightarrow \text{bool}$

$\langle \text{proof} \rangle$

**interpretation** *multiset-multiset-linorder*: *linorder*

*le-mset-mset* ::  $('a :: \text{linorder}) \text{multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{multiset multiset} \Rightarrow \text{bool}$

*less-mset-mset* ::  $('a :: \text{linorder}) \text{multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{multiset multiset} \Rightarrow \text{bool}$

$\langle \text{proof} \rangle$

**lemma** *wf-less-mset-mset*:  $\text{wf } \{(\Sigma :: ('a :: \text{wellorder}) \text{multiset multiset}, T). \Sigma \#<_{\#} T\}$

$\langle \text{proof} \rangle$

**interpretation** *multiset-multiset-wellorder*: *wellorder*

*le-mset-mset* ::  $('a :: \text{wellorder}) \text{multiset multiset} \Rightarrow ('a :: \text{wellorder}) \text{multiset multiset} \Rightarrow \text{bool}$

*less-mset-mset* ::  $('a :: \text{wellorder}) \text{multiset multiset} \Rightarrow ('a :: \text{wellorder}) \text{multiset multiset} \Rightarrow \text{bool}$

$\langle \text{proof} \rangle$

**lemma** *union-less-mset-mset-mono2*:  $B \#<_{\#} D \implies C + B \#<_{\#} C + (D :: 'a :: \text{order multiset multiset})$

$\langle \text{proof} \rangle$

**lemma** *union-less-mset-mset-diff-plus*:

$U \leq_{\#} \Sigma \implies T \#<_{\#} U \implies \Sigma - U + T \#<_{\#} \Sigma$

$\langle \text{proof} \rangle$

**lemma** *ex-gt-imp-less-mset-mset*:

$(\exists y :: 'a :: \text{linorder multiset} \in_{\#} T. (\forall x. x \in_{\#} \Sigma \longrightarrow x \#<_{\#} y)) \implies \Sigma \#<_{\#} T$

$\langle \text{proof} \rangle$

### 1.1.5 Remove

**lemma** *set-mset-minus-replicate-mset[simp]*:

$n \geq \text{count } A \ a \implies \text{set-mset } (A - \text{replicate-mset } n \ a) = \text{set-mset } A - \{a\}$   
 $n < \text{count } A \ a \implies \text{set-mset } (A - \text{replicate-mset } n \ a) = \text{set-mset } A$   
 $\langle \text{proof} \rangle$

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

**lemma** *mset-removeAll[simp, code]*:

*removeAll-mset* C (mset L) = mset (removeAll C L)  
 $\langle \text{proof} \rangle$

**lemma** *removeAll-mset-filter-mset*:

*removeAll-mset* C M = filter-mset (op  $\neq$  C) M  
 $\langle \text{proof} \rangle$

**abbreviation** *remove1-mset* :: 'a  $\Rightarrow$  'a multiset  $\Rightarrow$  'a multiset **where**  
*remove1-mset* C M  $\equiv$  M - {#C#}

**lemma** *remove1-mset-remove1[code]*:

*remove1-mset* C (mset L) = mset (remove1 C L)  
 $\langle \text{proof} \rangle$

**lemma** *in-remove1-mset-neg*:

**assumes** ab: a  $\neq$  b  
**shows** a  $\in \#$  remove1-mset b C  $\longleftrightarrow$  a  $\in \#$  C  
 $\langle \text{proof} \rangle$

**lemma** *size-mset-removeAll-mset-le-iff*:

size (removeAll-mset x M) < size M  $\longleftrightarrow$  x  $\in \#$  M  
 $\langle \text{proof} \rangle$

**lemma** *size-mset-remove1-mset-le-iff*:

size (remove1-mset x M) < size M  $\longleftrightarrow$  x  $\in \#$  M  
 $\langle \text{proof} \rangle$

**lemma** *set-mset-remove1-mset[simp]*:

set-mset (remove1-mset L (mset W)) = set (remove1 L W)  
 $\langle \text{proof} \rangle$

### 1.1.6 Replicate

**lemma** *replicate-mset-plus*: replicate-mset (a + b) C = replicate-mset a C + replicate-mset b C  
 $\langle \text{proof} \rangle$

**lemma** *mset-replicate-replicate-mset*:

mset (replicate n L) = replicate-mset n L  
 $\langle \text{proof} \rangle$

**lemma** *set-mset-single-iff-replicate-mset*:

set-mset U = {a}  $\longleftrightarrow$  ( $\exists n > 0. U = \text{replicate-mset } n \ a$ ) (**is** ?S  $\longleftrightarrow$  ?R)  
 $\langle \text{proof} \rangle$

### 1.1.7 Multiset and set conversion

**lemma** *count-mset-set-if*:

$\text{count } (\text{mset-set } A) \ a = (\text{if } a \in A \wedge \text{finite } A \text{ then } 1 \text{ else } 0)$   
 $\langle \text{proof} \rangle$

**lemma** *mset-set-set-mset-empty-mempty*[*iff*]:

$\text{mset-set } (\text{set-mset } D) = \{\#\} \longleftrightarrow D = \{\#\}$   
 $\langle \text{proof} \rangle$

**lemma** *size-mset-set-card*:

$\text{finite } S \implies \text{size } (\text{mset-set } S) = \text{card } S$   
 $\langle \text{proof} \rangle$

**lemma** *count-mset-set-le-one*:  $\text{count } (\text{mset-set } A) \ x \leq 1$

$\langle \text{proof} \rangle$

**lemma** *mset-set-subseteq-mset-set*[*iff*]:

**assumes**  $\text{finite } A \ \text{finite } B$   
**shows**  $\text{mset-set } A \subseteq\# \text{ mset-set } B \longleftrightarrow A \subseteq B$   
 $\langle \text{proof} \rangle$

**lemma** *mset-set-set-mset-subseteq*[*simp*]:  $\text{mset-set } (\text{set-mset } A) \subseteq\# A$

$\langle \text{proof} \rangle$

**lemma** *mset-sorted-list-of-set*[*simp*]:

$\text{mset } (\text{sorted-list-of-set } A) = \text{mset-set } A$   
 $\langle \text{proof} \rangle$

**lemma** *mset-take-subseteq*:  $\text{mset } (\text{take } n \ xs) \subseteq\# \text{ mset } xs$

$\langle \text{proof} \rangle$

### 1.1.8 Removing duplicates

**definition** *remdups-mset* ::  $'v \text{ multiset} \Rightarrow 'v \text{ multiset}$  **where**

$\text{remdups-mset } S = \text{mset-set } (\text{set-mset } S)$

**lemma** *remdups-mset-in*[*iff*]:  $a \in\# \text{ remdups-mset } A \longleftrightarrow a \in\# A$

$\langle \text{proof} \rangle$

**lemma** *count-remdups-mset-eq-1*:  $a \in\# \text{ remdups-mset } A \longleftrightarrow \text{count } (\text{remdups-mset } A) \ a = 1$

$\langle \text{proof} \rangle$

**lemma** *remdups-mset-empty*[*simp*]:

$\text{remdups-mset } \{\#\} = \{\#\}$   
 $\langle \text{proof} \rangle$

**lemma** *remdups-mset-singleton*[*simp*]:

$\text{remdups-mset } \{\#a\# \} = \{\#a\# \}$   
 $\langle \text{proof} \rangle$

**lemma** *set-mset-remdups*[*simp*]:  $\text{set-mset } (\text{remdups-mset } C) = \text{set-mset } C$

$\langle \text{proof} \rangle$

**lemma** *remdups-mset-eq-empty*[*iff*]:

$\text{remdups-mset } D = \{\#\} \longleftrightarrow D = \{\#\}$

$\langle \text{proof} \rangle$

**lemma** *remdups-mset-singleton-sum[simp]*:

$\text{remdups-mset } (\{\#a\# \} + A) = (\text{if } a \in \# A \text{ then } \text{remdups-mset } A \text{ else } \{\#a\# \} + \text{remdups-mset } A)$   
 $\text{remdups-mset } (A + \{\#a\# \}) = (\text{if } a \in \# A \text{ then } \text{remdups-mset } A \text{ else } \{\#a\# \} + \text{remdups-mset } A)$   
 $\langle \text{proof} \rangle$

**lemma** *mset-remdups-remdups-mset[simp]*:

$\text{mset } (\text{remdups } D) = \text{remdups-mset } (\text{mset } D)$   
 $\langle \text{proof} \rangle$

**definition** *distinct-mset* :: 'a multiset  $\Rightarrow$  bool **where**

$\text{distinct-mset } S \longleftrightarrow (\forall a. a \in \# S \longrightarrow \text{count } S a = 1)$

**lemma** *distinct-mset-empty[simp]*:  $\text{distinct-mset } \{\#\}$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-singleton[simp]*:  $\text{distinct-mset } \{\#a\#\}$

$\langle \text{proof} \rangle$

**definition** *distinct-mset-set* :: 'a multiset set  $\Rightarrow$  bool **where**

$\text{distinct-mset-set } \Sigma \longleftrightarrow (\forall S \in \Sigma. \text{distinct-mset } S)$

**lemma** *distinct-mset-set-empty[simp]*:

$\text{distinct-mset-set } \{\}$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-set-singleton[iff]*:

$\text{distinct-mset-set } \{A\} \longleftrightarrow \text{distinct-mset } A$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-set-insert[iff]*:

$\text{distinct-mset-set } (\text{insert } S \Sigma) \longleftrightarrow (\text{distinct-mset } S \wedge \text{distinct-mset-set } \Sigma)$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-set-union[iff]*:

$\text{distinct-mset-set } (\Sigma \cup \Sigma') \longleftrightarrow (\text{distinct-mset-set } \Sigma \wedge \text{distinct-mset-set } \Sigma')$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-union*:

**assumes** *dist*:  $\text{distinct-mset } (A + B)$

**shows**  $\text{distinct-mset } A$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-minus[simp]*:

$\text{distinct-mset } A \Longrightarrow \text{distinct-mset } (A - B)$

$\langle \text{proof} \rangle$

**lemma** *in-distinct-mset-set-distinct-mset*:

$a \in \Sigma \Longrightarrow \text{distinct-mset-set } \Sigma \Longrightarrow \text{distinct-mset } a$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-remdups-mset[simp]*:  $\text{distinct-mset } (\text{remdups-mset } S)$

$\langle \text{proof} \rangle$

**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**  $A \# \cup B = \text{remdups-mset } (A + B)$   
 ⟨proof⟩

**lemma** *distinct-mset-set-distinct*:  
*distinct-mset-set* (*mset* ' *set* *Cs*)  $\longleftrightarrow (\forall c \in \text{set } Cs. \text{distinct } c)$   
 ⟨proof⟩

**lemma** *distinct-mset-add-single*:  
*distinct-mset* ( $\{\#a\# \} + L$ )  $\longleftrightarrow \text{distinct-mset } L \wedge a \notin \# L$   
 ⟨proof⟩

**lemma** *distinct-mset-single-add*:  
*distinct-mset* ( $L + \{\#a\# \}$ )  $\longleftrightarrow \text{distinct-mset } L \wedge a \notin \# L$   
 ⟨proof⟩

**lemma** *distinct-mset-size-eq-card*:  
*distinct-mset* *C*  $\implies \text{size } C = \text{card } (\text{set-mset } C)$   
 ⟨proof⟩

Another characterisation of *distinct-mset*

**lemma** *distinct-mset-count-less-1*:  
*distinct-mset* *S*  $\longleftrightarrow (\forall a. \text{count } S \ a \leq 1)$   
 ⟨proof⟩

**lemma** *distinct-mset-add*:  
*distinct-mset* ( $L + L'$ )  $\longleftrightarrow \text{distinct-mset } L \wedge \text{distinct-mset } L' \wedge L \# \cap L' = \{\#\}$  (**is** ?*A*  $\longleftrightarrow$  ?*B*)  
 ⟨proof⟩

**lemma** *distinct-mset-set-mset-ident[simp]*: *distinct-mset* *M*  $\implies \text{mset-set } (\text{set-mset } M) = M$   
 ⟨proof⟩

**lemma** *distinct-finite-set-mset-subseteq-iff[iff]*:  
**assumes** *dist*: *distinct-mset* *M* **and** *fin*: *finite* *N*  
**shows**  $\text{set-mset } M \subseteq N \longleftrightarrow M \subseteq \# \text{mset-set } N$   
 ⟨proof⟩

**lemma** *distinct-mem-diff-mset*:  
**assumes** *dist*: *distinct-mset* *M* **and** *mem*:  $x \in \text{set-mset } (M - N)$   
**shows**  $x \notin \text{set-mset } N$   
 ⟨proof⟩

**lemma** *distinct-set-mset-eq*:  
**assumes**  
*dist-m*: *distinct-mset* *M* **and**  
*dist-n*: *distinct-mset* *N* **and**  
*set-eq*:  $\text{set-mset } M = \text{set-mset } N$   
**shows**  $M = N$

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-union-mset*:

**assumes**

*distinct-mset D* **and**

*distinct-mset C*

**shows** *distinct-mset (D # $\cup$  C)*

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-inter-mset*:

**assumes**

*distinct-mset D* **and**

*distinct-mset C*

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

$\langle \text{proof} \rangle$

**lemma** *distinct-mset-remove1-All*:

*distinct-mset C  $\implies$  remove1-mset L C = removeAll-mset L C*

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

### 1.1.9 Filter

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

$\langle \text{proof} \rangle$

**lemma** *image-mset-subseteq-mono*: *A  $\subseteq\#$  B  $\implies$  image-mset f A  $\subseteq\#$  image-mset f B*

$\langle \text{proof} \rangle$

**lemma** *image-filter-ne-mset[simp]*:

*image-mset f {#x  $\in\#$  M. f x  $\neq$  y#} = removeAll-mset y (image-mset f M)*

$\langle \text{proof} \rangle$

**lemma** *comprehension-mset-False[simp]*:

*{# L  $\in\#$  A. False#} = {#}*

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

**lemma** *filter-mset-union-mset*:

*filter-mset P (A # $\cup$  B) = filter-mset P A # $\cup$  filter-mset P B*

$\langle \text{proof} \rangle$

**lemma** *filter-mset-mset-set*:

*finite A  $\implies$  filter-mset P (mset-set A) = mset-set {a  $\in$  A. P a}*

$\langle \text{proof} \rangle$

### 1.1.10 Sums

**lemma** *msetsum-distrib[simp]*:

**fixes**  $C\ D :: 'a \Rightarrow 'b :: \{comm-monoid-add\}$   
**shows**  $(\sum_{x \in \#A}. C\ x + D\ x) = (\sum_{x \in \#A}. C\ x) + (\sum_{x \in \#A}. D\ x)$   
 $\langle proof \rangle$

**lemma** *msetsum-union-disjoint*:

**assumes**  $A \# \cap B = \{\#\}$   
**shows**  $(\sum_{La \in \#A \# \cup B}. f\ La) = (\sum_{La \in \#A}. f\ La) + (\sum_{La \in \#B}. f\ La)$   
 $\langle proof \rangle$

### 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*  $\Rightarrow$  '*a* *multiset*  $\Rightarrow$  *bool* **where**  
 $M' < M \longleftrightarrow M' \# \subset \# M$

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

**instance**

$\langle proof \rangle$

**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*:

$r^{++}\ a\ b \Longrightarrow r \leq s \Longrightarrow s^{++}\ a\ b$   
 $\langle proof \rangle$

**lemma** *tranclp-mono*:

**assumes** *mono*:  $r \leq s$   
**shows**  $r^{++} \leq s^{++}$   
 $\langle proof \rangle$

**lemma** *tranclp-idemp-rel*:

$R^{++++}\ a\ b \longleftrightarrow R^{++}\ a\ b$   
 $\langle proof \rangle$

Equivalent of the theorem *rtranclp-idemp*

**lemma** *trancl-idemp*:  $(r^+)^+ = r^+$   
 $\langle \text{proof} \rangle$

**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*:  $rtranclp\ r\ a\ b \longleftrightarrow (a = b \vee tranclp\ r\ a\ b)$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-unfold-end*:  $tranclp\ r\ a\ b \longleftrightarrow (\exists a'. rtranclp\ r\ a\ a' \wedge r\ a'\ b)$   
 $\langle \text{proof} \rangle$

Near duplicate of theorem *tranclpD*:

**lemma** *tranclp-unfold-begin*:  $tranclp\ r\ a\ b \longleftrightarrow (\exists a'. r\ a\ a' \wedge rtranclp\ r\ a'\ b)$   
 $\langle \text{proof} \rangle$

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

**lemma** *tranclp-rtranclp-rtranclp-rel*:  $R^{+++}\ a\ b \longleftrightarrow R^{**}\ a\ b$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-rtranclp-rtranclp[simp]*:  $R^{+++} = R^{**}$   
 $\langle \text{proof} \rangle$

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

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

## 1.2.2 Full Transitions

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

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

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

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

We define output notations only for printing:

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

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

**lemma** *rtranclp-full1I*:  
 $R^{**}\ a\ b \Longrightarrow \text{full1}\ R\ b\ c \Longrightarrow \text{full1}\ R\ a\ c$   
 $\langle \text{proof} \rangle$



**lemma** *trancplp-full1I:*

$R^{++} a b \implies full1 R b c \implies full1 R a c$   
 $\langle proof \rangle$

**lemma** *rtrancplp-fullI:*

$R^{**} a b \implies full R b c \implies full R a c$   
 $\langle proof \rangle$

**lemma** *trancplp-full-full1I:*

$R^{++} a b \implies full R b c \implies full1 R a c$   
 $\langle proof \rangle$

**lemma** *full-fullI:*

$R a b \implies full R b c \implies full1 R a c$   
 $\langle proof \rangle$

**lemma** *full-unfold:*

$full r S S' \longleftrightarrow ((S = S' \wedge no\text{-}step r S') \vee full1 r S S')$   
 $\langle proof \rangle$

**lemma** *full1-is-full[intro]:*  $full1 R S T \implies full R S T$

$\langle proof \rangle$

**lemma** *not-full1-rtrancplp-relation:*  $\neg full1 R^{**} a b$

$\langle proof \rangle$

**lemma** *not-full-rtrancplp-relation:*  $\neg full R^{**} a b$

$\langle proof \rangle$

**lemma** *full1-trancplp-relation-full:*

$full1 R^{++} a b \longleftrightarrow full1 R a b$   
 $\langle proof \rangle$

**lemma** *full-trancplp-relation-full:*

$full R^{++} a b \longleftrightarrow full R a b$   
 $\langle proof \rangle$

**lemma** *rtrancplp-full1-eq-or-full1:*

$(full1 R)^{**} a b \longleftrightarrow (a = b \vee full1 R a b)$   
 $\langle proof \rangle$

**lemma** *trancplp-full1-full1:*

$(full1 R)^{++} a b \longleftrightarrow full1 R a b$   
 $\langle proof \rangle$

### 1.2.3 Well-Foundedness and Full Transitions

**lemma** *wf-exists-normal-form:*

**assumes**  $wf: wf \ \{(x, y). R \ y \ x\}$   
**shows**  $\exists b. R^{**} a b \wedge no\text{-}step R b$   
 $\langle proof \rangle$

**lemma** *wf-exists-normal-form-full:*

**assumes**  $wf: wf \ \{(x, y). R \ y \ x\}$   
**shows**  $\exists b. full R a b$

$\langle proof \rangle$

#### 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*:

$wf\ R \implies (\bigwedge a\ b. (a, b) \in S \implies (\nu\ a, \nu\ b) \in R) \implies wf\ S$

$\langle proof \rangle$

**lemma** *wfP-if-measure*: **fixes**  $f :: 'a \Rightarrow nat$

**shows**  $(\bigwedge x\ y. P\ x \implies g\ x\ y \implies f\ y < f\ x) \implies wf\ \{(y, x). P\ x \wedge g\ x\ y\}$

$\langle proof \rangle$

**lemma** *wf-if-measure-f*:

**assumes**  $wf\ r$

**shows**  $wf\ \{(b, a). (f\ b, f\ a) \in r\}$

$\langle proof \rangle$

**lemma** *wf-wf-if-measure'*:

**assumes**  $wf\ r$  **and**  $H: \bigwedge x\ y. P\ x \implies g\ x\ y \implies (f\ y, f\ x) \in r$

**shows**  $wf\ \{(y, x). P\ x \wedge g\ x\ y\}$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *wfP-if-measure2*: **fixes**  $f :: 'a \Rightarrow nat$

**shows**  $(\bigwedge x\ y. P\ x\ y \implies g\ x\ y \implies f\ x < f\ y) \implies wf\ \{(x, y). P\ x\ y \wedge g\ x\ y\}$

$\langle proof \rangle$

**lemma** *lexord-on-finite-set-is-wf*:

**assumes**

$P\text{-finite}: \bigwedge U. P\ U \longrightarrow U \in A$  **and**

$finite: finite\ A$  **and**

$wf: wf\ R$  **and**

$trans: trans\ R$

**shows**  $wf\ \{(T, S). (P\ S \wedge P\ T) \wedge (T, S) \in lexord\ R\}$

$\langle proof \rangle$

**lemma** *wf-fst-wf-pair*:

**assumes**  $wf\ \{(M', M). R\ M'\ M\}$

**shows**  $wf\ \{((M', N'), (M, N)). R\ M'\ M\}$

$\langle proof \rangle$

**lemma** *wf-snd-wf-pair*:

**assumes**  $wf\ \{(M', M). R\ M'\ M\}$

**shows**  $wf\ \{((M', N'), (M, N)). R\ N'\ N\}$

$\langle proof \rangle$

**lemma** *wf-if-measure-f-notation2*:

**assumes**  $wf\ r$

**shows**  $wf \{(b, h\ a) \mid b\ a. (f\ b, f\ (h\ a)) \in r\}$   
 $\langle proof \rangle$

**lemma** *wf-wf-if-measure'-notation2*:

**assumes**  $wf\ r$  **and**  $H: \bigwedge x\ y. P\ x \implies g\ x\ y \implies (f\ y, f\ (h\ x)) \in r$

**shows**  $wf \{(y, h\ x) \mid y\ x. P\ x \wedge g\ x\ y\}$

$\langle proof \rangle$

**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*  $((\bigwedge n. \forall m < n. ?P\ m \implies ?P\ n) \implies ?P\ ?n)$ , 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**

$\bigwedge n. (\forall m < Suc\ n. P\ m) \implies P\ (Suc\ n)$

**shows**  $P\ n$

$\langle proof \rangle$

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* $[simp]$ :

$0 < (if\ P\ then\ a\ else\ (0::nat)) \longleftrightarrow P \wedge 0 < a$

$\langle proof \rangle$

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**  $\exists n. f\ n > m \wedge n > n_0$

$\langle proof \rangle$

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**  $\text{bounded } f \longleftrightarrow \text{bounded } (\lambda i. k * f i)$   
 $\langle \text{proof} \rangle$

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

**lemma** *bounded-finite-linorder*:  
**fixes**  $f :: 'a \Rightarrow 'a :: \{\text{finite}, \text{linorder}\}$   
**shows**  $\text{bounded } f$   
 $\langle \text{proof} \rangle$

## 1.4 More List

### 1.4.1 *upt*

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

**lemma** *upt-Suc-le-append*:  $\neg i \leq j \implies [i..<\text{Suc } j] = []$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *take-upt-bound-minus[simp]*:  
**assumes**  $i \leq n - m$   
**shows**  $\text{take } i [m..<n] = [m ..<m+i]$   
 $\langle \text{proof} \rangle$

**lemma** *append-cons-eq-upt*:  
**assumes**  $A @ B = [m..<n]$   
**shows**  $A = [m ..<m+\text{length } A]$  and  $B = [m + \text{length } A..<n]$   
 $\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

A more restrictive version holds:

**lemma**  $B \neq [] \implies A @ B = [m..<n] \longleftrightarrow A = [m ..<m+\text{length } A] \wedge B = [m + \text{length } A..<n]$   
**(is**  $?P \implies ?A = ?B$   
 $\langle \text{proof} \rangle$

**lemma** *append-cons-eq-upt-length-i*:  
**assumes**  $A @ i \# B = [m..<n]$   
**shows**  $A = [m ..<i]$   
 $\langle \text{proof} \rangle$

**lemma** *append-cons-eq-upt-length*:

**assumes**  $A @ i \# B = [m..<n]$

**shows**  $\text{length } A = i - m$

$\langle \text{proof} \rangle$

**lemma** *append-cons-eq-upt-length-i-end*:

**assumes**  $A @ i \# B = [m..<n]$

**shows**  $B = [\text{Suc } i ..<n]$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *upt-decomp-lt*:

**assumes**  $H: xs @ i \# ys @ j \# zs = [m ..< n]$

**shows**  $i < j$

$\langle \text{proof} \rangle$

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*:

$\text{length } W = \text{Suc } 0 \longleftrightarrow (\exists L. W = [L])$

$\langle \text{proof} \rangle$

**lemma** *length-list-2*:  $\text{length } S = 2 \longleftrightarrow (\exists a \ b. S = [a, b])$

$\langle \text{proof} \rangle$

**lemma** *finite-bounded-list*:

**fixes**  $b :: \text{nat}$

**shows**  $\text{finite } \{xs. \text{length } xs < s \wedge (\forall i < \text{length } xs. xs ! i < b)\} \text{ (is finite } (?S \ s))$

$\langle \text{proof} \rangle$

## 1.4.2 Lexicographic Ordering

**lemma** *lexn-Suc*:

$(x \# xs, y \# ys) \in \text{lexn } r \ (\text{Suc } n) \longleftrightarrow$

$(\text{length } xs = n \wedge \text{length } ys = n) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in \text{lexn } r \ n))$

$\langle \text{proof} \rangle$

**lemma** *lexn-n*:

$n > 0 \implies (x \# xs, y \# ys) \in \text{lexn } r \ n \longleftrightarrow$

$(\text{length } xs = n-1 \wedge \text{length } ys = n-1) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in \text{lexn } r \ (n-1)))$

$\langle \text{proof} \rangle$

There is some subtle point in the proof here.  $1$  is converted to  $\text{Suc } 0$ , 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*:

$([a, b], [c, d]) \in \text{lexn } r \ 2 \longleftrightarrow (a, c) \in r \vee (a = c \wedge (b, d) \in r)$

$\langle \text{proof} \rangle$

**lemma** *lexn3-conv*:

$([a, b, c], [a', b', c']) \in \text{lexn } r \ 3 \longleftrightarrow$

$(a, a') \in r \vee (a = a' \wedge (b, b') \in r) \vee (a = a' \wedge b = b' \wedge (c, c') \in r)$

$\langle \text{proof} \rangle$

### 1.4.3 Remove

#### More lemmas about remove

**lemma** *remove1-Nil*:

$remove1\ (-\ L)\ W = [] \longleftrightarrow (W = [] \vee W = [-L])$   
 $\langle proof \rangle$

**lemma** *remove1-mset-single-add*:

$a \neq b \implies remove1\ mset\ a\ (\{ \#b \# \} + C) = \{ \#b \# \} + remove1\ mset\ a\ C$   
 $remove1\ mset\ a\ (\{ \#a \# \} + C) = C$   
 $\langle proof \rangle$

#### 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*  $x\ xs = remove1\ cond\ ((op =)\ x)\ xs$

$\langle proof \rangle$

**lemma** *mset-map-mset-remove1-cond*:

$mset\ (map\ mset\ (remove1\ cond\ (\lambda L. mset\ L = mset\ a)\ C)) =$   
 $remove1\ mset\ (mset\ a)\ (mset\ (map\ mset\ C))$   
 $\langle proof \rangle$

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*  $x\ xs = removeAll\ cond\ ((op =)\ x)\ xs$

$\langle proof \rangle$

**lemma** *removeAll-cond*  $P\ xs = filter\ (\lambda x. \neg P\ x)\ xs$

$\langle proof \rangle$

**lemma** *mset-map-mset-removeAll-cond*:

$mset\ (map\ mset\ (removeAll\ cond\ (\lambda b. mset\ b = mset\ a)\ C)) =$   
 $= removeAll\ mset\ (mset\ a)\ (mset\ (map\ mset\ C))$   
 $\langle proof \rangle$

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

**abbreviation** *union-mset-list* **where**

$union\ mset\ list\ xs\ ys \equiv case\ prod\ append\ (fold\ (\lambda x\ (ys,\ zs). (remove1\ x\ ys,\ x \# zs))\ xs\ (ys,\ []))$

**lemma** *union-mset-list*:

$mset\ xs \# \cup\ mset\ ys = mset\ (union\ mset\ list\ xs\ ys)$   
 $\langle proof \rangle$

## Filter

**lemma** *distinct-filter-eq-if:*

*distinct C  $\implies$  length (filter (op = L) C) = (if L  $\in$  set C then 1 else 0)*  
 *$\langle$ proof $\rangle$*

**end**





## Chapter 2

# Definition of Entailment

This chapter defines various form of entailment.

**end**

### 2.1 Clausal Logic

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

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

#### 2.1.1 Literals

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

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

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

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

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

**lemma** *ex-lit-cases*:  $(\exists L. P\ L) \longleftrightarrow (\exists A. P\ (Pos\ A) \vee P\ (Neg\ A))$   
*<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]:  $- \text{Pos } A = \text{Neg } A$  and*  
*uminus-Neg[simp]:  $- \text{Neg } A = \text{Pos } A$*   
*⟨proof⟩*

**lemma** *atm-of-uminus[simp]:*

*atm-of  $(-L) = \text{atm-of } L$*   
*⟨proof⟩*

**lemma** *uminus-of-uminus-id[simp]:*

*$- (- (x :: 'v \text{ literal})) = x$*   
*⟨proof⟩*

**lemma** *uminus-not-id[simp]:*

*$x \neq - (x :: 'v \text{ literal})$*   
*⟨proof⟩*

**lemma** *uminus-not-id'[simp]:*

*$- x \neq (x :: 'v \text{ literal})$*   
*⟨proof⟩*

**lemma** *uminus-eq-inj[iff]:*

*$-(a :: 'v \text{ literal}) = -b \iff a = b$*   
*⟨proof⟩*

**lemma** *uminus-lit-swap:*

*$(a :: 'a \text{ literal}) = -b \iff -a = b$*   
*⟨proof⟩*

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

**begin**

**definition** *less-literal :: 'a literal  $\Rightarrow$  'a literal  $\Rightarrow$  bool where*

*less-literal  $L M \iff \text{atm-of } L < \text{atm-of } M \vee \text{atm-of } L \leq \text{atm-of } M \wedge \text{is-neg } L < \text{is-neg } M$*

**definition** *less-eq-literal :: 'a literal  $\Rightarrow$  'a literal  $\Rightarrow$  bool where*

*less-eq-literal  $L M \iff \text{atm-of } L < \text{atm-of } M \vee \text{atm-of } L \leq \text{atm-of } M \wedge \text{is-neg } L \leq \text{is-neg } M$*

**instance**

*⟨proof⟩*

**end**

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

**begin**

**instance**

*⟨proof⟩*

**end**

**lemma** *pos-less-neg[simp]:  $\text{Pos } A < \text{Neg } A$*

*⟨proof⟩*

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

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

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

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

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

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

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

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

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

**lemma** *leq-imp-less-eq-atm-of*:  $L \leq M \implies atm-of\ L \leq atm-of\ M$   
 $\langle proof \rangle$

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

**instance**  
 $\langle proof \rangle$

**end**

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

**instance**  
 $\langle proof \rangle$

**end**

## 2.1.2 Clauses

Clauses are (finite) multisets of literals.

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

**abbreviation** *poss* :: *'a multiset*  $\Rightarrow$  *'a clause* **where** *poss* *AA*  $\equiv \{\#Pos\ A.\ A \in \# AA\}$

**abbreviation** *negs* :: *'a multiset*  $\Rightarrow$  *'a clause* **where** *negs* *AA*  $\equiv \{\#Neg\ A.\ A \in \# AA\}$

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

$\langle \text{proof} \rangle$

**lemma** *Max-in-lits*:  $C \neq \{\#\} \implies \text{Max} (\text{set-mset } C) \in\# C$

$\langle \text{proof} \rangle$

**lemma** *Max-atm-of-set-mset-commute*:  $C \neq \{\#\} \implies \text{Max} (\text{atm-of } \text{' set-mset } C) = \text{atm-of } (\text{Max} (\text{set-mset } C))$

$\langle \text{proof} \rangle$

**lemma** *Max-pos-neg-less-multiset*:

**assumes** *max*:  $\text{Max} (\text{set-mset } C) = \text{Pos } A$  **and** *neg*:  $\text{Neg } A \in\# D$

**shows**  $C \# \subseteq\# D$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**definition** *atms-of* :: 'a clause  $\Rightarrow$  'a set **where**

*atms-of*  $C = \text{atm-of } \text{' set-mset } C$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *atms-of-union-mset[simp]*:

$\text{atms-of } (A \# \cup B) = \text{atms-of } A \cup \text{atms-of } B$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *atm-of-lit-in-atms-of*:  $L \in\# C \implies \text{atm-of } L \in \text{atms-of } C$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *pos-lit-in-atms-of*:  $\text{Pos } A \in\# C \implies A \in \text{atms-of } C$

$\langle \text{proof} \rangle$

**lemma** *neg-lit-in-atms-of*:  $\text{Neg } A \in\# C \implies A \in \text{atms-of } C$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *atm-of-eq-atm-of*:

$\text{atm-of } L = \text{atm-of } L' \longleftrightarrow (L = L' \vee L = -L')$

$\langle \text{proof} \rangle$

**lemma** *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set:*

*atm-of*  $L \in \text{atm-of } I \iff (L \in I \vee -L \in I)$

$\langle \text{proof} \rangle$

**lemma** *lits-subseteq-imp-atms-subseteq:*  $\text{set-mset } C \subseteq \text{set-mset } D \implies \text{atms-of } C \subseteq \text{atms-of } D$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma**

*atms-of-poss[simp]:*  $\text{atms-of } (\text{poss } AA) = \text{set-mset } AA$  **and**

*atms-of-negg[simp]:*  $\text{atms-of } (\text{negs } AA) = \text{set-mset } AA$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *le-multiset-Max-in-imp-Max:*

$\text{Max } (\text{atms-of } D) = A \implies C \# \subseteq \# D \implies A \in \text{atms-of } C \implies \text{Max } (\text{atms-of } C) = A$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *Max-lit-eq-pos-or-neg-Max-atm:*

$C \neq \{\#\} \implies \text{Max } (\text{set-mset } C) = \text{Pos } (\text{Max } (\text{atms-of } C)) \vee \text{Max } (\text{set-mset } C) = \text{Neg } (\text{Max } (\text{atms-of } C))$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**end**

## 2.2 Clausal Logic

**theory** *Herbrand-Interpretation*

**imports** *Clausal-Logic*

**begin**

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

### 2.2.1 Herbrand Interpretations

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

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

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

**lemma** *true-lit-simps[simp]*:  
 $I \models_l \text{Pos } A \longleftrightarrow A \in I$   
 $I \models_l \text{Neg } A \longleftrightarrow A \notin I$   
 $\langle \text{proof} \rangle$

**lemma** *true-lit-iff[iff]*:  $I \models_l L \longleftrightarrow (\exists A. L = \text{Pos } A \wedge A \in I \vee L = \text{Neg } A \wedge A \notin I)$   
 $\langle \text{proof} \rangle$

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

**lemma** *true-cls-empty[iff]*:  $\neg I \models \{\#\}$   
 $\langle \text{proof} \rangle$

**lemma** *true-cls-singleton[iff]*:  $I \models \{\#L\# \} \longleftrightarrow I \models_l L$   
 $\langle \text{proof} \rangle$

**lemma** *true-cls-union[iff]*:  $I \models C + D \longleftrightarrow I \models C \vee I \models D$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *true-cls-replicate-mset[iff]*:  $I \models \text{replicate-mset } n L \longleftrightarrow n \neq 0 \wedge I \models_l L$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *pos-neg-in-imp-true*:  $\text{Pos } A \in \# C \Longrightarrow \text{Neg } A \in \# C \Longrightarrow I \models C$   
 $\langle \text{proof} \rangle$

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

**lemma** *true-clss-empty[iff]*:  $I \models_s \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *true-clss-singleton[iff]*:  $I \models_s \{C\} \longleftrightarrow I \models C$   
 $\langle \text{proof} \rangle$

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

$\langle proof \rangle$

**lemma** *true-clss-mono*:  $DD \subseteq CC \Rightarrow I \models_s CC \Rightarrow I \models_s DD$

$\langle proof \rangle$

**abbreviation** *satisfiable* :: 'a clause set  $\Rightarrow$  bool **where**

*satisfiable*  $CC \equiv \exists I. I \models_s CC$

**definition** *true-cls-mset* :: 'a interp  $\Rightarrow$  'a clause multiset  $\Rightarrow$  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*]:  $I \models_m \{\#\}$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *true-cls-mset-union*[*iff*]:  $I \models_m CC + DD \longleftrightarrow I \models_m CC \wedge I \models_m DD$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *true-cls-mset-mono*:  $\text{set-mset } DD \subseteq \text{set-mset } CC \Rightarrow I \models_m CC \Rightarrow I \models_m DD$

$\langle proof \rangle$

**lemma** *true-clss-set-mset*[*iff*]:  $I \models_s \text{set-mset } CC \longleftrightarrow I \models_m CC$

$\langle proof \rangle$

**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  $\Rightarrow$  'a literal  $\Rightarrow$  bool (**infix**  $\models_l$  50) **where**

$I \models_l L \longleftrightarrow L \in I$

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

## Consistency

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

**lemma** *consistent-interp-empty*[*simp*]:  
*consistent-interp*  $\{\}$   $\langle$ proof $\rangle$

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

**lemma** *consistent-interp-subset*:

**assumes**

$A \subseteq B$  **and**

*consistent-interp*  $B$

**shows** *consistent-interp*  $A$

$\langle$ proof $\rangle$

**lemma** *consistent-interp-change-insert*:

$a \notin A \Rightarrow \neg a \notin A \Rightarrow \text{consistent-interp } (\text{insert } (\neg a) A) \longleftrightarrow \text{consistent-interp } (\text{insert } a A)$   
 $\langle$ proof $\rangle$

**lemma** *consistent-interp-insert-pos*[*simp*]:

$a \notin A \Rightarrow \text{consistent-interp } (\text{insert } a A) \longleftrightarrow \text{consistent-interp } A \wedge \neg a \notin A$   
 $\langle$ proof $\rangle$

**lemma** *consistent-interp-insert-not-in*:

*consistent-interp*  $A \Rightarrow a \notin A \Rightarrow \neg a \notin A \Rightarrow \text{consistent-interp } (\text{insert } a A)$   
 $\langle$ proof $\rangle$

## Atoms

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

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

**lemma** *atms-of-mmultiset*[*simp*]:

*atms-of* (*mset*  $a$ ) = *atm-of* ' *set*  $a$   
 $\langle$ proof $\rangle$

**lemma** *atms-of-ms-mset-unfold*:

*atms-of-ms* (*mset* '  $b$ ) =  $(\bigcup_{x \in b. \text{atm-of } ' \text{set } x}$   
 $\langle$ proof $\rangle$

**definition** *atms-of-s* :: 'a literal set  $\Rightarrow$  'a set **where**

*atms-of-s*  $C = \text{atm-of } ' C$

**lemma** *atms-of-ms-empty-set*[*simp*]:

*atms-of-ms*  $\{\} = \{\}$   
 $\langle$ proof $\rangle$



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

**lemma** *atms-of-ms-mono*:  
 $A \subseteq B \implies atms-of-ms A \subseteq atms-of-ms B$   
 $\langle proof \rangle$

**lemma** *atms-of-ms-finite*[simp]:  
 $finite \psi s \implies finite (atms-of-ms \psi s)$   
 $\langle proof \rangle$

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

**lemma** *atms-of-ms-insert*[simp]:  
 $atms-of-ms (insert \psi s \chi s) = atms-of \psi s \cup atms-of-ms \chi s$   
 $\langle proof \rangle$

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

**lemma** *atms-of-atms-of-ms-mono*[simp]:  
 $A \in \psi \implies atms-of A \subseteq atms-of-ms \psi$   
 $\langle proof \rangle$

**lemma** *atms-of-ms-single-set-mset-atms-of*[simp]:  
 $atms-of-ms (single \text{ ' } set-mset B) = atms-of B$   
 $\langle proof \rangle$

**lemma** *atms-of-ms-remove-incl*:  
**shows**  $atms-of-ms (Set.remove a \psi) \subseteq atms-of-ms \psi$   
 $\langle proof \rangle$

**lemma** *atms-of-ms-remove-subset*:  
 $atms-of-ms (\varphi - \psi) \subseteq atms-of-ms \varphi$   
 $\langle proof \rangle$

**lemma** *finite-atms-of-ms-remove-subset*[simp]:  
 $finite (atms-of-ms A) \implies finite (atms-of-ms (A - C))$   
 $\langle proof \rangle$

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

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

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

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

**lemma** *atms-of-s-union[simp]*:  
 $\text{atms-of-s } (Ia \cup Ib) = \text{atms-of-s } Ia \cup \text{atms-of-s } Ib$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *in-atms-of-s-decomp[iff]*:  
 $P \in \text{atms-of-s } I \longleftrightarrow (\text{Pos } P \in I \vee \text{Neg } P \in I) \text{ (is } ?P \longleftrightarrow ?Q)$   
 $\langle \text{proof} \rangle$

**lemma** *atm-of-in-atm-of-set-in-uminus*:  
 $\text{atm-of } L' \in \text{atm-of } 'B \implies L' \in B \vee -L' \in B$   
 $\langle \text{proof} \rangle$

## Totality

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

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

**lemma** *total-over-set-empty[simp]*:  
 $\text{total-over-set } I \text{ } \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *total-over-m-empty[simp]*:  
 $\text{total-over-m } I \text{ } \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *total-over-set-single[iff]*:  
 $\text{total-over-set } I \text{ } \{L\} \longleftrightarrow (\text{Pos } L \in I \vee \text{Neg } L \in I)$   
 $\langle \text{proof} \rangle$

**lemma** *total-over-set-insert[iff]*:  
 $\text{total-over-set } I \text{ } (\text{insert } L \text{ } Ls) \longleftrightarrow ((\text{Pos } L \in I \vee \text{Neg } L \in I) \wedge \text{total-over-set } I \text{ } Ls)$   
 $\langle \text{proof} \rangle$

**lemma** *total-over-set-union[iff]*:  
 $\text{total-over-set } I \text{ } (Ls \cup Ls') \longleftrightarrow (\text{total-over-set } I \text{ } Ls \wedge \text{total-over-set } I \text{ } Ls')$   
 $\langle \text{proof} \rangle$

**lemma** *total-over-m-subset*:  
 $A \subseteq B \implies \text{total-over-m } I \text{ } B \implies \text{total-over-m } I \text{ } A$

$\langle \text{proof} \rangle$

**lemma** *total-over-m-sum*[iff]:

**shows**  $\text{total-over-m } I \{C + D\} \longleftrightarrow (\text{total-over-m } I \{C\} \wedge \text{total-over-m } I \{D\})$

$\langle \text{proof} \rangle$

**lemma** *total-over-m-union*[iff]:

$\text{total-over-m } I (A \cup B) \longleftrightarrow (\text{total-over-m } I A \wedge \text{total-over-m } I B)$

$\langle \text{proof} \rangle$

**lemma** *total-over-m-insert*[iff]:

$\text{total-over-m } I (\text{insert } a \ A) \longleftrightarrow (\text{total-over-set } I (\text{atms-of } a) \wedge \text{total-over-m } I A)$

$\langle \text{proof} \rangle$

**lemma** *total-over-m-extension*:

**fixes**  $I :: 'v \text{ literal set}$  **and**  $A :: 'v \text{ clauses}$

**assumes**  $\text{total}: \text{total-over-m } I A$

**shows**  $\exists I'. \text{total-over-m } (I \cup I') (A \cup B)$

$\wedge (\forall x \in I'. \text{atm-of } x \in \text{atms-of-ms } B \wedge \text{atm-of } x \notin \text{atms-of-ms } A)$

$\langle \text{proof} \rangle$

**lemma** *total-over-m-consistent-extension*:

**fixes**  $I :: 'v \text{ literal set}$  **and**  $A :: 'v \text{ clauses}$

**assumes**

$\text{total}: \text{total-over-m } I A$  **and**

$\text{cons}: \text{consistent-interp } I$

**shows**  $\exists I'. \text{total-over-m } (I \cup I') (A \cup B)$

$\wedge (\forall x \in I'. \text{atm-of } x \in \text{atms-of-ms } B \wedge \text{atm-of } x \notin \text{atms-of-ms } A) \wedge \text{consistent-interp } (I \cup I')$

$\langle \text{proof} \rangle$

**lemma** *total-over-set-atms-of-m*[simp]:

$\text{total-over-set } Ia (\text{atms-of-s } Ia)$

$\langle \text{proof} \rangle$

**lemma** *total-over-set-literal-defined*:

**assumes**  $\{\#A\# \} + D \in \psi s$

**and**  $\text{total-over-set } I (\text{atms-of-ms } \psi s)$

**shows**  $A \in I \vee -A \in I$

$\langle \text{proof} \rangle$

**lemma** *tot-over-m-remove*:

**assumes**  $\text{total-over-m } (I \cup \{L\}) \{\psi\}$

**and**  $L: L \notin \# \ \psi \ -L \notin \# \ \psi$

**shows**  $\text{total-over-m } I \{\psi\}$

$\langle \text{proof} \rangle$

**lemma** *total-union*:

**assumes**  $\text{total-over-m } I \psi$

**shows**  $\text{total-over-m } (I \cup I') \psi$

$\langle \text{proof} \rangle$

**lemma** *total-union-2*:

**assumes**  $\text{total-over-m } I \psi$

**and**  $\text{total-over-m } I' \psi'$

**shows**  $\text{total-over-m } (I \cup I') (\psi \cup \psi')$

$\langle \text{proof} \rangle$

## Interpretations

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

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

**lemma** *true-cls-singleton*[iff]:  $I \models \{\#L\# \} \longleftrightarrow I \models_l L$   
 $\langle proof \rangle$

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

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

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

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

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

**lemma** *true-cls-replicate-mset*[iff]:  $I \models replicate-mset\ n\ L \longleftrightarrow n \neq 0 \wedge I \models_l L$   
 $\langle proof \rangle$

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

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

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

**lemma** *true-clss-empty*[simp]:  $I \models_s \{\}$   
 $\langle proof \rangle$

**lemma** *true-clss-singleton*[iff]:  $I \models_s \{C\} \longleftrightarrow I \models C$   
 $\langle proof \rangle$

**lemma** *true-clss-empty-entails-empty*[iff]:  $\{\} \models_s N \longleftrightarrow N = \{\}$   
 $\langle proof \rangle$

**lemma** *true-cls-insert-l* [simp]:  
 $M \models A \implies \text{insert } L \ M \models A$   
 ⟨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]:  $I \models_s \text{insert } C \ DD \longleftrightarrow I \models C \wedge I \models_s DD$   
 ⟨proof⟩

**lemma** *true-clss-mono*:  $DD \subseteq CC \implies I \models_s CC \implies I \models_s DD$   
 ⟨proof⟩

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

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

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

**lemma** *model-remove*[simp]:  $I \models_s N \implies I \models_s \text{Set.remove } a \ N$   
 ⟨proof⟩

**lemma** *model-remove-minus*[simp]:  $I \models_s N \implies I \models_s N - A$   
 ⟨proof⟩

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

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

## Satisfiability

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

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

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

**lemma** *satisfiable-decreasing*:  
**assumes** *satisfiable*  $(\psi \cup \psi')$   
**shows** *satisfiable*  $\psi$   
 $\langle$ *proof* $\rangle$

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

## Entailment for Multisets of Clauses

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

**lemma** *true-cls-mset-empty[simp]*:  $I \models_m \{\#\}$   
 $\langle$ *proof* $\rangle$

**lemma** *true-cls-mset-singleton[iff]*:  $I \models_m \{\#C\# \} \longleftrightarrow I \models C$   
 $\langle$ *proof* $\rangle$

**lemma** *true-cls-mset-union[iff]*:  $I \models_m CC + DD \longleftrightarrow I \models_m CC \wedge I \models_m DD$   
 $\langle$ *proof* $\rangle$

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

**lemma** *true-cls-mset-mono*:  $\text{set-mset } DD \subseteq \text{set-mset } CC \Longrightarrow I \models_m CC \Longrightarrow I \models_m DD$   
 $\langle$ *proof* $\rangle$

**lemma** *true-clss-set-mset[iff]*:  $I \models_s \text{set-mset } CC \longleftrightarrow I \models_m CC$   
 $\langle$ *proof* $\rangle$

**lemma** *true-cls-mset-increasing-r[simp]*:  
 $I \models_m CC \Longrightarrow I \cup J \models_m CC$   
 $\langle$ *proof* $\rangle$

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

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

A simple application of the previous theorem:

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

$\langle proof \rangle$

**lemma** *multiset-not-empty*:

**assumes**  $M \neq \{\#\}$

**and**  $x \in \# M$

**shows**  $\exists A. x = Pos A \vee x = Neg A$

$\langle proof \rangle$

**lemma** *atms-of-ms-empty*:

**fixes**  $\psi :: 'v \text{ clauses}$

**assumes**  $atms-of-ms \psi = \{\}$

**shows**  $\psi = \{\} \vee \psi = \{\{\#\}\}$

$\langle proof \rangle$

**lemma** *consistent-interp-disjoint*:

**assumes**  $consI: consistent-interp I$

**and**  $disj: atms-of-s A \cap atms-of-s I = \{\}$

**and**  $consA: consistent-interp A$

**shows**  $consistent-interp (A \cup I)$

$\langle proof \rangle$

**lemma** *total-remove-unused*:

**assumes**  $total-over-m I \psi$

**shows**  $total-over-m \{v \in I. atm-of v \in atms-of-ms \psi\} \psi$

$\langle proof \rangle$

**lemma** *true-cls-remove-hd-if-notin-vars*:

**assumes**  $insert a M' \models D$

**and**  $atm-of a \notin atms-of D$

**shows**  $M' \models D$

$\langle proof \rangle$

**lemma** *total-over-set-atm-of*:

**fixes**  $I :: 'v \text{ interp}$  **and**  $K :: 'v \text{ set}$

**shows**  $total-over-set I K \longleftrightarrow (\forall l \in K. l \in (atm-of ' I))$

$\langle proof \rangle$

## Tautologies

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

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

**lemma** *tautology-Pos-Neg[intro]*:

**assumes**  $Pos p \in \# A$  **and**  $Neg p \in \# A$

**shows**  $tautology A$

$\langle proof \rangle$

**lemma** *tautology-minus[simp]*:

**assumes**  $L \in \# A$  **and**  $-L \in \# A$

**shows**  $tautology A$

$\langle proof \rangle$

**lemma** *tautology-exists-Pos-Neg*:

**assumes**  $tautology \psi$

**shows**  $\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi$   
 $\langle \text{proof} \rangle$

**lemma** *tautology-decomp*:  
 $\text{tautology } \psi \longleftrightarrow (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$   
 $\langle \text{proof} \rangle$

**lemma** *tautology-false[simp]*:  $\neg \text{tautology } \{ \# \}$   
 $\langle \text{proof} \rangle$

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

**lemma** *minus-interp-tautology*:  
**assumes**  $\{ -L \mid L. L \in \# \chi \} \models \chi$   
**shows**  $\text{tautology } \chi$   
 $\langle \text{proof} \rangle$

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

**lemma** *tautology-imp-tautology*:  
**fixes**  $\chi \chi' :: 'a \text{ clause}$   
**assumes**  $\forall I. \text{total-over-m } I \{ \chi \} \longrightarrow I \models \chi \longrightarrow I \models \chi' \text{ and } \text{tautology } \chi$   
**shows**  $\text{tautology } \chi' \langle \text{proof} \rangle$

## Entailment for clauses and propositions

We also need entailment of clauses by other clauses.

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

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

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

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

**lemma** *true-cls-cls-refl[simp]*:  
 $A \models_f A$   
 $\langle \text{proof} \rangle$

**lemma** *true-cls-cls-insert-l[simp]*:  
 $a \models_f C \implies \text{insert } a \ A \models_p C$   
 $\langle \text{proof} \rangle$

**lemma** *true-cls-clss-empty[iff]*:  
 $N \models_{fs} \{ \}$



$\langle proof \rangle$

**lemma** *true-prop-true-clause*[iff]:

$\{\varphi\} \models_p \psi \longleftrightarrow \varphi \models_f \psi$

$\langle proof \rangle$

**lemma** *true-clss-clss-true-clss-cl*[iff]:

$N \models_{ps} \{\psi\} \longleftrightarrow N \models_p \psi$

$\langle proof \rangle$

**lemma** *true-clss-clss-true-clss-clss*[iff]:

$\{\chi\} \models_{ps} \psi \longleftrightarrow \chi \models_{fs} \psi$

$\langle proof \rangle$

**lemma** *true-clss-clss-empty*[simp]:

$N \models_{ps} \{\}$

$\langle proof \rangle$

**lemma** *true-clss-cl-subset*:

$A \subseteq B \implies A \models_p CC \implies B \models_p CC$

$\langle proof \rangle$

**lemma** *true-clss-cl-mono-l*[simp]:

$A \models_p CC \implies A \cup B \models_p CC$

$\langle proof \rangle$

**lemma** *true-clss-cl-mono-l2*[simp]:

$B \models_p CC \implies A \cup B \models_p CC$

$\langle proof \rangle$

**lemma** *true-clss-cl-mono-r*[simp]:

$A \models_p CC \implies A \models_p CC + CC'$

$\langle proof \rangle$

**lemma** *true-clss-cl-mono-r'*[simp]:

$A \models_p CC' \implies A \models_p CC + CC'$

$\langle proof \rangle$

**lemma** *true-clss-clss-union-l*[simp]:

$A \models_{ps} CC \implies A \cup B \models_{ps} CC$

$\langle proof \rangle$

**lemma** *true-clss-clss-union-l-r*[simp]:

$B \models_{ps} CC \implies A \cup B \models_{ps} CC$

$\langle proof \rangle$

**lemma** *true-clss-cl-in*[simp]:

$CC \in A \implies A \models_p CC$

$\langle proof \rangle$

**lemma** *true-clss-cl-insert-l*[simp]:

$A \models_p C \implies \text{insert } a \ A \models_p C$

$\langle proof \rangle$

**lemma** *true-clss-clss-insert-l*[simp]:

$A \models_{ps} C \implies \text{insert } a \ A \models_{ps} C$

$\langle proof \rangle$

**lemma** *true-clss-clss-union-and*[iff]:

$A \models_{ps} C \cup D \longleftrightarrow (A \models_{ps} C \wedge A \models_{ps} D)$

$\langle proof \rangle$

**lemma** *true-clss-clss-insert*[iff]:

$A \models_{ps} insert\ L\ Ls \longleftrightarrow (A \models_p L \wedge A \models_{ps} Ls)$

$\langle proof \rangle$

**lemma** *true-clss-clss-subset*:

$A \subseteq B \implies A \models_{ps} CC \implies B \models_{ps} CC$

$\langle proof \rangle$

**lemma** *union-trus-clss-clss*[simp]:  $A \cup B \models_{ps} B$

$\langle proof \rangle$

**lemma** *true-clss-clss-remove*[simp]:

$A \models_{ps} B \implies A \models_{ps} B - C$

$\langle proof \rangle$

**lemma** *true-clss-clss-subsetE*:

$N \models_{ps} B \implies A \subseteq B \implies N \models_{ps} A$

$\langle proof \rangle$

**lemma** *true-clss-clss-in-imp-true-clss-clss*:

**assumes**  $N \models_{ps} U$

**and**  $A \in U$

**shows**  $N \models_p A$

$\langle proof \rangle$

**lemma** *all-in-true-clss-clss*:  $\forall x \in B. x \in A \implies A \models_{ps} B$

$\langle proof \rangle$

**lemma** *true-clss-clss-left-right*:

**assumes**  $A \models_{ps} B$

**and**  $A \cup B \models_{ps} M$

**shows**  $A \models_{ps} M \cup B$

$\langle proof \rangle$

**lemma** *true-clss-clss-generalise-true-clss-clss*:

$A \cup C \models_{ps} D \implies B \models_{ps} C \implies A \cup B \models_{ps} D$

$\langle proof \rangle$

**lemma** *true-clss-clss-or-true-clss-clss-or-not-true-clss-clss-or*:

**assumes**  $D: N \models_p D + \{\# - L\#\}$

**and**  $C: N \models_p C + \{\#L\#\}$

**shows**  $N \models_p D + C$

$\langle proof \rangle$

**lemma** *true-clss-clss-union-mset*[iff]:  $I \models C \# \cup D \longleftrightarrow I \models C \vee I \models D$

$\langle proof \rangle$

**lemma** *true-clss-clss-union-mset-true-clss-clss-or-not-true-clss-clss-or*:

**assumes**

$D: N \models_p D + \{\# - L\#\}$  **and**

$C: N \models_p C + \{\#L\# \}$   
**shows**  $N \models_p D \# \cup C$   
 $\langle \text{proof} \rangle$

**lemma** *satisfiable-carac*[iff]:

$(\exists I. \text{consistent-interp } I \wedge I \models_s \varphi) \longleftrightarrow \text{satisfiable } \varphi$  (**is**  $(\exists I. ?Q I) \longleftrightarrow ?S$ )  
 $\langle \text{proof} \rangle$

**lemma** *satisfiable-carac'*[simp]:  $\text{consistent-interp } I \implies I \models_s \varphi \implies \text{satisfiable } \varphi$   
 $\langle \text{proof} \rangle$

### 2.3.3 Subsumptions

**lemma** *subsumption-total-over-m*:

**assumes**  $A \subseteq \# B$   
**shows**  $\text{total-over-m } I \{B\} \implies \text{total-over-m } I \{A\}$   
 $\langle \text{proof} \rangle$

**lemma** *atms-of-replicate-mset-replicate-mset-uminus*[simp]:

$\text{atms-of } (D - \text{replicate-mset } (\text{count } D L) L - \text{replicate-mset } (\text{count } D (-L)) (-L))$   
 $= \text{atms-of } D - \{\text{atm-of } L\}$   
 $\langle \text{proof} \rangle$

**lemma** *subsumption-chained*:

**assumes**  
 $\forall I. \text{total-over-m } I \{D\} \longrightarrow I \models D \longrightarrow I \models \varphi$  **and**  
 $C \subseteq \# D$   
**shows**  $(\forall I. \text{total-over-m } I \{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \vee \text{tautology } \varphi$   
 $\langle \text{proof} \rangle$

### 2.3.4 Removing Duplicates

**lemma** *tautology-remdups-mset*[iff]:

$\text{tautology } (\text{remdups-mset } C) \longleftrightarrow \text{tautology } C$   
 $\langle \text{proof} \rangle$

**lemma** *atms-of-remdups-mset*[simp]:  $\text{atms-of } (\text{remdups-mset } C) = \text{atms-of } C$   
 $\langle \text{proof} \rangle$

**lemma** *true-cls-remdups-mset*[iff]:  $I \models \text{remdups-mset } C \longleftrightarrow I \models C$   
 $\langle \text{proof} \rangle$

**lemma** *true-clss-cls-remdups-mset*[iff]:  $A \models_p \text{remdups-mset } C \longleftrightarrow A \models_p C$   
 $\langle \text{proof} \rangle$

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

**lemma** *simple-clss-empty*[*simp*]:  
*simple-clss* {} = {{#}}  
 <proof>

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

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

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

**lemma** *cls-in-simple-clss*:  
**shows** {#}  $\in$  *simple-clss* *s*  
 <proof>

**lemma** *simple-clss-card*:  
**fixes** *atms* :: 'v set  
**assumes** *finite* *atms*  
**shows** *card* (*simple-clss* *atms*)  $\leq$  (*3::nat*)  $\wedge$  (*card* *atms*)  
 <proof>

**lemma** *simple-clss-mono*:  
**assumes** *incl*: *atms*  $\subseteq$  *atms'*  
**shows** *simple-clss* *atms*  $\subseteq$  *simple-clss* *atms'*  
 <proof>

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

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

### 2.3.6 Experiment: Expressing the Entailments as Locales

```

locale entail =
  fixes entail :: 'a set  $\Rightarrow$  'b  $\Rightarrow$  bool (infix  $\models_e$  50)
  assumes entail-insert[simp]:  $I \neq \{\} \implies \text{insert } L \ I \models_e x \longleftrightarrow \{L\} \models_e x \vee I \models_e x$ 
  assumes entail-union[simp]:  $I \models_e A \implies I \cup I' \models_e A$ 
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]:
   $I \models_{es} \{\}$ 
   $\langle \text{proof} \rangle$ 

lemma entails-single[iff]:
   $I \models_{es} \{a\} \longleftrightarrow I \models_e a$ 
   $\langle \text{proof} \rangle$ 

lemma entails-insert-l[simp]:
   $M \models_{es} A \implies \text{insert } L \ M \models_{es} A$ 
   $\langle \text{proof} \rangle$ 

lemma entails-union[iff]:  $I \models_{es} CC \cup DD \longleftrightarrow I \models_{es} CC \wedge I \models_{es} DD$ 
   $\langle \text{proof} \rangle$ 

lemma entails-insert[iff]:  $I \models_{es} \text{insert } C \ DD \longleftrightarrow I \models_e C \wedge I \models_{es} DD$ 
   $\langle \text{proof} \rangle$ 

lemma entails-insert-mono:  $DD \subseteq CC \implies I \models_{es} CC \implies I \models_{es} DD$ 
   $\langle \text{proof} \rangle$ 

lemma entails-union-increase[simp]:
  assumes  $I \models_{es} \psi$ 
  shows  $I \cup I' \models_{es} \psi$ 
   $\langle \text{proof} \rangle$ 

lemma true-clss-commute-l:
   $I \cup I' \models_{es} \psi \longleftrightarrow I' \cup I \models_{es} \psi$ 
   $\langle \text{proof} \rangle$ 

lemma entails-remove[simp]:  $I \models_{es} N \implies I \models_{es} \text{Set.remove } a \ N$ 
   $\langle \text{proof} \rangle$ 

lemma entails-remove-minus[simp]:  $I \models_{es} N \implies I \models_{es} N - A$ 
   $\langle \text{proof} \rangle$ 

end

interpretation true-cls: entail true-cls
   $\langle \text{proof} \rangle$ 

```

### 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  $\Rightarrow$  'a literal multiset set  $\Rightarrow$  bool (**infix**  $\models_{\text{sext}}$  49)

**where**

$I \models_{\text{sext}} N \iff (\forall J. I \subseteq J \longrightarrow \text{consistent-interp } J \longrightarrow \text{total-over-m } J \ N \longrightarrow J \models_s N)$

**lemma** *true-clss-imp-true-clss-ext*:

$I \models_s N \implies I \models_{\text{sext}} N$

$\langle \text{proof} \rangle$

**lemma** *true-clss-ext-decrease-right-remove-r*:

**assumes**  $I \models_{\text{sext}} N$

**shows**  $I \models_{\text{sext}} N - \{C\}$

$\langle \text{proof} \rangle$

**lemma** *consistent-true-clss-ext-satisfiable*:

**assumes** *consistent-interp*  $I$  **and**  $I \models_{\text{sext}} A$

**shows** *satisfiable*  $A$

$\langle \text{proof} \rangle$

**lemma** *not-consistent-true-clss-ext*:

**assumes**  $\neg \text{consistent-interp } I$

**shows**  $I \models_{\text{sext}} A$

$\langle \text{proof} \rangle$

**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*  $\equiv \{CF\} \cup \{CT\} \cup \{CVar\ x \mid x. True\}$

**definition** *binary-connectives*  $\equiv \{CAnd, COr, CImp, CEq\}$

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

**lemma** *propo-induct-arity*[*case-names nullary unary binary*]:

**fixes**  $\varphi\ \psi :: 'v\ propo$   
 **assumes** *nullary*:  $\bigwedge \varphi\ x. \varphi = FF \vee \varphi = FT \vee \varphi = FVar\ x \implies P\ \varphi$   
 **and** *unary*:  $\bigwedge \psi. P\ \psi \implies P\ (FNot\ \psi)$   
 **and** *binary*:  $\bigwedge \varphi\ \psi1\ \psi2. P\ \psi1 \implies P\ \psi2 \implies \varphi = FAnd\ \psi1\ \psi2 \vee \varphi = FOR\ \psi1\ \psi2 \vee \varphi = FImp\ \psi1\ \psi2$   
  $\vee \varphi = FEq\ \psi1\ \psi2 \implies P\ \varphi$   
 **shows**  $P\ \psi$   
 *<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  $\Rightarrow$  'v propo list  $\Rightarrow$  'v propo where
conn CT [] = FT |
conn CF [] = FF |
conn (CVar v) [] = FVar v |
conn CNot [ $\varphi$ ] = FNot  $\varphi$  |
conn CAnd ( $\varphi$  # [ $\psi$ ]) = FAnd  $\varphi$   $\psi$  |
conn COr ( $\varphi$  # [ $\psi$ ]) = FOr  $\varphi$   $\psi$  |
conn CImp ( $\varphi$  # [ $\psi$ ]) = FImp  $\varphi$   $\psi$  |
conn CEq ( $\varphi$  # [ $\psi$ ]) = FEq  $\varphi$   $\psi$  |
conn - - = FF
```

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

```
lemma connective-cases-arity[case-names nullary binary unary]:
assumes nullary:  $\bigwedge x. c = CT \vee c = CF \vee c = CVar x \implies P$ 
and binary:  $c \in \text{binary-connectives} \implies P$ 
and unary:  $c = CNot \implies P$ 
shows P
<proof>
```

```
lemma connective-cases-arity-2[case-names nullary unary binary]:
assumes nullary:  $c \in \text{nullary-connective} \implies P$ 
and unary:  $c = CNot \implies P$ 
and binary:  $c \in \text{binary-connectives} \implies P$ 
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  $\Rightarrow$  'v propo list  $\Rightarrow$  bool for c :: 'v connective where
wf-conn-nullary[simp]:  $(c = CT \vee c = CF \vee c = CVar v) \implies \text{wf-conn } c []$  |
wf-conn-unary[simp]:  $c = CNot \implies \text{wf-conn } c [\psi]$  |
wf-conn-binary[simp]:  $c \in \text{binary-connectives} \implies \text{wf-conn } c (\psi \# \psi' \# [])$ 
thm wf-conn.induct
```

```
lemma wf-conn-induct[consumes 1, case-names CT CF CVar CNot COr CAnd CImp CEq]:
assumes wf-conn c x and
 $\bigwedge v. c = CT \implies P []$  and
 $\bigwedge v. c = CF \implies P []$  and
 $\bigwedge v. c = CVar v \implies P []$  and
 $\bigwedge \psi. c = CNot \implies P [\psi]$  and
 $\bigwedge \psi \psi'. c = COr \implies P [\psi, \psi']$  and
 $\bigwedge \psi \psi'. c = CAnd \implies P [\psi, \psi']$  and
 $\bigwedge \psi \psi'. c = CImp \implies P [\psi, \psi']$  and
 $\bigwedge \psi \psi'. c = CEq \implies P [\psi, \psi']$ 
shows P x
<proof>
```

### 3.1.2 properties of the abstraction

First we can define simplification rules.

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



$wf\text{-}conn\ CT\ l \implies conn\ CT\ l = FT$   
 $wf\text{-}conn\ CF\ l \implies conn\ CF\ l = FF$   
 $wf\text{-}conn\ (CVar\ x)\ l \implies conn\ (CVar\ x)\ l = FVar\ x$   
 $\langle proof \rangle$

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

$wf\text{-}conn\ CT\ l \longleftrightarrow l = []$   
 $wf\text{-}conn\ CF\ l \longleftrightarrow l = []$   
 $wf\text{-}conn\ (CVar\ x)\ l \longleftrightarrow l = []$   
 $wf\text{-}conn\ CNot\ (\xi @ \varphi \# \xi') \longleftrightarrow \xi = [] \wedge \xi' = []$   
 $\langle proof \rangle$

**lemma** *wf-conn-list*:

$wf\text{-}conn\ c\ l \implies conn\ c\ l = FT \longleftrightarrow (c = CT \wedge l = [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FF \longleftrightarrow (c = CF \wedge l = [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FVar\ x \longleftrightarrow (c = CVar\ x \wedge l = [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FAnd\ a\ b \longleftrightarrow (c = CAnd \wedge l = a \# b \# [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FOr\ a\ b \longleftrightarrow (c = COr \wedge l = a \# b \# [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FEq\ a\ b \longleftrightarrow (c = CEq \wedge l = a \# b \# [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FImp\ a\ b \longleftrightarrow (c = CImp \wedge l = a \# b \# [])$   
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FNot\ a \longleftrightarrow (c = CNot \wedge l = a \# [])$   
 $\langle proof \rangle$

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

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

$\langle proof \rangle$

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

**lemma** *wf-conn-bin-list-length*:

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

**lemma** *wf-conn-not-list-length[iff]*:

**fixes**  $l :: 'v\ propo\ list$   
**shows**  $wf\text{-}conn\ CNot\ l \longleftrightarrow length\ l = 1$   
 $\langle proof \rangle$

Decomposing the Not into an element is moreover very useful.

**lemma** *wf-conn-Not-decomp*:

**fixes**  $l :: 'v\ propo\ list$  **and**  $a :: 'v$   
**assumes**  $corr: wf\text{-}conn\ CNot\ l$   
**shows**  $\exists\ a.\ l = [a]$   
 $\langle proof \rangle$

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*:

$length\ l = length\ l' \implies wf\text{-}conn\ c\ l \longleftrightarrow wf\text{-}conn\ c\ l'$   
 $\langle proof \rangle$

**lemma** *wf-conn-no-arity-change-helper*:  
 $\text{length } (\xi @ \varphi \# \xi') = \text{length } (\xi @ \varphi' \# \xi')$   
 $\langle \text{proof} \rangle$

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

**lemma** *conn-inj-not*:  
**assumes** *correct*: *wf-conn* *c* *l*  
**and** *conn*: *conn* *c* *l* = *FNot*  $\psi$   
**shows** *c* = *CNot* **and** *l* = [ $\psi$ ]  
 $\langle \text{proof} \rangle$

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

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

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*  $\varphi \preceq \psi \Rightarrow \varphi \preceq \psi$   
 $\langle \text{proof} \rangle$

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

**lemma** *subformula-trans*:  
 $\psi \preceq \psi' \Rightarrow \varphi \preceq \psi \Rightarrow \varphi \preceq \psi'$   
 $\langle \text{proof} \rangle$

**lemma** *subformula-leaf*:  
**fixes**  $\varphi \ \psi$  :: '*v* *propo*  
**assumes** *incl*:  $\varphi \preceq \psi$   
**and** *simple*:  $\psi = \text{FT} \vee \psi = \text{FF} \vee \psi = \text{FVar } x$   
**shows**  $\varphi = \psi$   
 $\langle \text{proof} \rangle$

**lemma** *subformula-not-incl-eq*:

**assumes**  $\varphi \preceq \text{conn } c \ l$   
**and**  $\text{wf-conn } c \ l$   
**and**  $\forall \psi. \psi \in \text{set } l \longrightarrow \neg \varphi \preceq \psi$   
**shows**  $\varphi = \text{conn } c \ l$   
 $\langle \text{proof} \rangle$

**lemma** *wf-subformula-conn-cases*:

$\text{wf-conn } c \ l \implies \varphi \preceq \text{conn } c \ l \longleftrightarrow (\varphi = \text{conn } c \ l \vee (\exists \psi. \psi \in \text{set } l \wedge \varphi \preceq \psi))$   
 $\langle \text{proof} \rangle$

**lemma** *subformula-decomp-explicit[simp]*:

$\varphi \preceq \text{FAnd } \psi \ \psi' \longleftrightarrow (\varphi = \text{FAnd } \psi \ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$  (**is**  $?P \text{ FAnd}$ )  
 $\varphi \preceq \text{FOr } \psi \ \psi' \longleftrightarrow (\varphi = \text{FOr } \psi \ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$   
 $\varphi \preceq \text{FEq } \psi \ \psi' \longleftrightarrow (\varphi = \text{FEq } \psi \ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$   
 $\varphi \preceq \text{FImp } \psi \ \psi' \longleftrightarrow (\varphi = \text{FImp } \psi \ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$   
 $\langle \text{proof} \rangle$

**lemma** *wf-conn-helper-facts[iff]*:

$\text{wf-conn } \text{CNot } [\varphi]$   
 $\text{wf-conn } \text{CT } []$   
 $\text{wf-conn } \text{CF } []$   
 $\text{wf-conn } (\text{CVar } x) []$   
 $\text{wf-conn } \text{CAnd } [\varphi, \psi]$   
 $\text{wf-conn } \text{COr } [\varphi, \psi]$   
 $\text{wf-conn } \text{CImp } [\varphi, \psi]$   
 $\text{wf-conn } \text{CEq } [\varphi, \psi]$   
 $\langle \text{proof} \rangle$

**lemma** *exists-c-conn*:  $\exists \ c \ l. \varphi = \text{conn } c \ l \wedge \text{wf-conn } c \ l$

$\langle \text{proof} \rangle$

**lemma** *subformula-conn-decomp[simp]*:

**assumes**  $\text{wf: wf-conn } c \ l$   
**shows**  $\varphi \preceq \text{conn } c \ l \longleftrightarrow (\varphi = \text{conn } c \ l \vee (\exists \psi \in \text{set } l. \varphi \preceq \psi))$  (**is**  $?A \longleftrightarrow ?B$ )  
 $\langle \text{proof} \rangle$

**lemma** *subformula-leaf-explicit[simp]*:

$\varphi \preceq \text{FT} \longleftrightarrow \varphi = \text{FT}$   
 $\varphi \preceq \text{FF} \longleftrightarrow \varphi = \text{FF}$   
 $\varphi \preceq \text{FVar } x \longleftrightarrow \varphi = \text{FVar } x$   
 $\langle \text{proof} \rangle$

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

**primrec** *vars-of-prop*::  $'v \text{ propo} \Rightarrow 'v \text{ set}$  **where**

$\text{vars-of-prop } \text{FT} = \{\}$  |  
 $\text{vars-of-prop } \text{FF} = \{\}$  |  
 $\text{vars-of-prop } (\text{FVar } x) = \{x\}$  |  
 $\text{vars-of-prop } (\text{FNot } \varphi) = \text{vars-of-prop } \varphi$  |  
 $\text{vars-of-prop } (\text{FAnd } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi$  |  
 $\text{vars-of-prop } (\text{FOr } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi$  |  
 $\text{vars-of-prop } (\text{FImp } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi$  |  
 $\text{vars-of-prop } (\text{FEq } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi$

**lemma** *vars-of-prop-incl-conn*:

**fixes**  $\xi \ \xi' :: 'v \text{ propo list}$  **and**  $\psi :: 'v \text{ propo}$  **and**  $c :: 'v \text{ connective}$   
**assumes**  $\text{corr: wf-conn } c \ l$  **and**  $\text{incl: } \psi \in \text{set } l$

**shows**  $\text{vars-of-prop } \psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$   
 $\langle \text{proof} \rangle$

The set of variables is compatible with the subformula order.

**lemma** *subformula-vars-of-prop*:

$\varphi \preceq \psi \implies \text{vars-of-prop } \varphi \subseteq \text{vars-of-prop } \psi$   
 $\langle \text{proof} \rangle$

### 3.1.4 Positions

Instead of 1 or 2 we use  $L$  or  $R$

**datatype**  $\text{sign} = L \mid R$

We use  $\text{nil}$  instead of  $\varepsilon$ .

**fun**  $\text{pos} :: 'v \text{ propo} \Rightarrow \text{sign list set}$  **where**

$\text{pos } FF = \{\square\} \mid$   
 $\text{pos } FT = \{\square\} \mid$   
 $\text{pos } (FVar \ x) = \{\square\} \mid$   
 $\text{pos } (FAnd \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$   
 $\text{pos } (FOr \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$   
 $\text{pos } (FEq \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$   
 $\text{pos } (FImp \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$   
 $\text{pos } (FNot \ \varphi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\}$

**lemma** *finite-pos*:  $\text{finite } (\text{pos } \varphi)$

$\langle \text{proof} \rangle$

**lemma** *finite-inj-comp-set*:

**fixes**  $s :: 'v \text{ set}$   
**assumes**  $\text{finite}: \text{finite } s$   
**and**  $\text{inj}: \text{inj } f$   
**shows**  $\text{card } (\{f \ p \mid p. p \in s\}) = \text{card } s$   
 $\langle \text{proof} \rangle$

**lemma** *cons-inject*:

$\text{inj } (op \ \# \ s)$   
 $\langle \text{proof} \rangle$

**lemma** *finite-insert-nil-cons*:

$\text{finite } s \implies \text{card } (\text{insert } \square \ \{L \ \# \ p \mid p. p \in s\}) = 1 + \text{card } \{L \ \# \ p \mid p. p \in s\}$   
 $\langle \text{proof} \rangle$

**lemma** *card-not[simp]*:

$\text{card } (\text{pos } (FNot \ \varphi)) = 1 + \text{card } (\text{pos } \varphi)$   
 $\langle \text{proof} \rangle$

**lemma** *card-seperate*:

**assumes**  $\text{finite } s1$  **and**  $\text{finite } s2$   
**shows**  $\text{card } (\{L \ \# \ p \mid p. p \in s1\} \cup \{R \ \# \ p \mid p. p \in s2\}) = \text{card } (\{L \ \# \ p \mid p. p \in s1\})$   
 $+ \text{card } (\{R \ \# \ p \mid p. p \in s2\})$  (**is**  $\text{card } (?L \cup ?R) = \text{card } ?L + \text{card } ?R$ )  
 $\langle \text{proof} \rangle$

**definition** *prop-size* **where**  $\text{prop-size } \varphi = \text{card } (\text{pos } \varphi)$

**lemma** *prop-size-vars-of-prop*:

**fixes**  $\varphi :: 'v \text{ propo}$

**shows**  $\text{card } (\text{vars-of-prop } \varphi) \leq \text{prop-size } \varphi$

$\langle \text{proof} \rangle$

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

**inductive** *path-to* :: *sign list*  $\Rightarrow$  *'v propo*  $\Rightarrow$  *'v propo*  $\Rightarrow$  *bool* **where**

*path-to-refl*[*intro*]: *path-to* []  $\varphi \varphi$  |

*path-to-l*:  $c \in \text{binary-connectives} \vee c = \text{CNot} \implies \text{wf-conn } c (\varphi \# l) \implies \text{path-to } p \varphi \varphi' \implies$

*path-to* (*L* # *p*) (*conn* *c* ( $\varphi \# l$ ))  $\varphi'$  |

*path-to-r*:  $c \in \text{binary-connectives} \implies \text{wf-conn } c (\psi \# \varphi \# []) \implies \text{path-to } p \varphi \varphi' \implies$

*path-to* (*R* # *p*) (*conn* *c* ( $\psi \# \varphi \# []$ ))  $\varphi'$

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*  $\varphi \varphi' \implies \varphi' \preceq \varphi$

$\langle \text{proof} \rangle$

**lemma** *subformula-path-exists*:

**fixes**  $\varphi \varphi' :: 'v \text{ propo}$

**shows**  $\varphi' \preceq \varphi \implies \exists p. \text{path-to } p \varphi \varphi'$

$\langle \text{proof} \rangle$

**fun** *replace-at* :: *sign list*  $\Rightarrow$  *'v propo*  $\Rightarrow$  *'v propo*  $\Rightarrow$  *'v propo* **where**

*replace-at* [] -  $\psi = \psi$  |

*replace-at* (*L* # *l*) (*FAnd*  $\varphi \varphi'$ )  $\psi = \text{FAnd } (\text{replace-at } l \varphi \psi) \varphi'$  |

*replace-at* (*R* # *l*) (*FAnd*  $\varphi \varphi'$ )  $\psi = \text{FAnd } \varphi (\text{replace-at } l \varphi' \psi)$  |

*replace-at* (*L* # *l*) (*FOr*  $\varphi \varphi'$ )  $\psi = \text{FOr } (\text{replace-at } l \varphi \psi) \varphi'$  |

*replace-at* (*R* # *l*) (*FOr*  $\varphi \varphi'$ )  $\psi = \text{FOr } \varphi (\text{replace-at } l \varphi' \psi)$  |

*replace-at* (*L* # *l*) (*FEq*  $\varphi \varphi'$ )  $\psi = \text{FEq } (\text{replace-at } l \varphi \psi) \varphi'$  |

*replace-at* (*R* # *l*) (*FEq*  $\varphi \varphi'$ )  $\psi = \text{FEq } \varphi (\text{replace-at } l \varphi' \psi)$  |

*replace-at* (*L* # *l*) (*FImp*  $\varphi \varphi'$ )  $\psi = \text{FImp } (\text{replace-at } l \varphi \psi) \varphi'$  |

*replace-at* (*R* # *l*) (*FImp*  $\varphi \varphi'$ )  $\psi = \text{FImp } \varphi (\text{replace-at } l \varphi' \psi)$  |

*replace-at* (*L* # *l*) (*FNot*  $\varphi$ )  $\psi = \text{FNot } (\text{replace-at } l \varphi \psi)$

## 3.2 Semantics over the syntax

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

**fun** *eval* :: (*'v*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'v propo*  $\Rightarrow$  *bool* (**infix**  $\models$  50) **where**

$\mathcal{A} \models \text{FT} = \text{True}$  |

$\mathcal{A} \models \text{FF} = \text{False}$  |

$\mathcal{A} \models \text{FVar } v = (\mathcal{A} \ v)$  |

$\mathcal{A} \models \text{FNot } \varphi = (\neg(\mathcal{A} \models \varphi))$  |

$\mathcal{A} \models \text{FAnd } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \wedge \mathcal{A} \models \varphi_2)$  |

$\mathcal{A} \models \text{FOr } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \vee \mathcal{A} \models \varphi_2)$  |

$\mathcal{A} \models \text{FImp } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2)$  |

$\mathcal{A} \models \text{FEq } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \longleftrightarrow \mathcal{A} \models \varphi_2)$

**definition** *evalf* (**infix**  $\models_f$  50) **where**

*evalf*  $\varphi \psi = (\forall A. A \models \varphi \longrightarrow A \models \psi)$

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

**theorem** *deduction-theorem*:

$\varphi \models^f \psi \longleftrightarrow (\forall A. A \models FImp \varphi \psi)$   
 $\langle proof \rangle$

A shorter proof:

**lemma**  $\varphi \models^f \psi \longleftrightarrow (\forall A. A \models FImp \varphi \psi)$   
 $\langle proof \rangle$

**definition** *same-over-set*::  $('v \Rightarrow bool) \Rightarrow ('v \Rightarrow bool) \Rightarrow 'v \text{ set} \Rightarrow bool$  **where**  
*same-over-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*  $\varphi$ )  
**shows**  $A \models \varphi \longleftrightarrow B \models \varphi$   
 $\langle proof \rangle$

**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 full formula: the relation  $r$  works on terms, while *propo-rew-step* works on formulas.

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

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

This lemma is only a health condition:

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

**shows**  $\text{propo-rew-step } r \ \varphi \ \varphi' \implies \exists \ \psi \ \psi'. \ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r \ \psi \ \psi'$   
 $\langle proof \rangle$

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

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

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

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

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

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

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

### 3.3.2 Consistency preservation

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

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

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

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

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

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

**lemma** *star-consistency-preservation*:  
 $\text{preserves-un-sat } r \implies \text{preserves-un-sat } (\text{propo-rew-step } r)^{\wedge **}$   
 $\langle \text{proof} \rangle$

### 3.3.3 Full Lifting

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

**lemma** *full-ropo-rew-step-preservers-val*[simp]:

*preserves-un-sat*  $r \implies \text{preserves-un-sat } (\text{full } (\text{propo-rew-step } r))$   
 $\langle \text{proof} \rangle$

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

*full* (*propo-rew-step*  $r$ )  $\varphi' \varphi \implies \neg(\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi')$   
 $\langle \text{proof} \rangle$

## 3.4 Transformation testing

### 3.4.1 Definition and first properties

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

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

**lemma** *test-symb-imp-all-subformula-st*[simp]:

*test-symb*  $FT \implies \text{all-subformula-st test-symb } FT$   
*test-symb*  $FF \implies \text{all-subformula-st test-symb } FF$   
*test-symb* (*FVar*  $x$ )  $\implies \text{all-subformula-st test-symb } (\text{FVar } x)$   
 $\langle \text{proof} \rangle$

**lemma** *all-subformula-st-test-symb-true-phi*:

*all-subformula-st test-symb*  $\varphi \implies \text{test-symb } \varphi$   
 $\langle \text{proof} \rangle$

**lemma** *all-subformula-st-decomp-imp*:

*wf-conn*  $c \ l \implies (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$   
 $\implies \text{all-subformula-st test-symb } (\text{conn } c \ l)$   
 $\langle \text{proof} \rangle$

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*  $c \ l$ )  $\implies \text{wf-conn } c \ l$   
 $\implies (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$   
 $\langle \text{proof} \rangle$

**lemma** *all-subformula-st-decomp*:

**fixes**  $c :: \text{'v connective}$  **and**  $l :: \text{'v propo list}$   
**assumes** *wf-conn*  $c \ l$   
**shows** *all-subformula-st test-symb* (*conn*  $c \ l$ )  
 $\longleftrightarrow (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$   
 $\langle \text{proof} \rangle$



**lemma** *helper-fact*:  $c \in \text{binary-connectives} \longleftrightarrow (c = COr \vee c = CAnd \vee c = CEq \vee c = CImp)$

*<proof>*

**lemma** *all-subformula-st-decomp-explicit[simp]*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$

**shows** *all-subformula-st test-symb* (FAnd  $\varphi \psi$ )

$\longleftrightarrow (\text{test-symb } (FAnd \varphi \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$

**and** *all-subformula-st test-symb* (FOr  $\varphi \psi$ )

$\longleftrightarrow (\text{test-symb } (FOr \varphi \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$

**and** *all-subformula-st test-symb* (FNot  $\varphi$ )

$\longleftrightarrow (\text{test-symb } (FNot \varphi) \wedge \text{all-subformula-st test-symb } \varphi)$

**and** *all-subformula-st test-symb* (FEq  $\varphi \psi$ )

$\longleftrightarrow (\text{test-symb } (FEq \varphi \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$

**and** *all-subformula-st test-symb* (FImp  $\varphi \psi$ )

$\longleftrightarrow (\text{test-symb } (FImp \varphi \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$

*<proof>*

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

**lemma** *subformula-all-subformula-st*:

$\psi \preceq \varphi \implies \text{all-subformula-st test-symb } \varphi \implies \text{all-subformula-st test-symb } \psi$

*<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  $\neg \text{all-subformula-st test-symb } \varphi$ , then something can be rewritten in  $\varphi$ .

**lemma** *no-test-symb-step-exists*:

**fixes**  $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  **and** *test-symb* ::  $'v \text{ propo} \Rightarrow \text{bool}$  **and**  $x :: 'v$

**and**  $\varphi :: 'v \text{ propo}$

**assumes**

*test-symb-false-nullary*:  $\forall x. \text{test-symb } FF \wedge \text{test-symb } FT \wedge \text{test-symb } (FVar \ x)$  **and**

$\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg \text{test-symb } \varphi') \longrightarrow (\exists \psi. r \ \varphi' \ \psi)$  **and**

$\neg \text{all-subformula-st test-symb } \varphi$

**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  $\forall \varphi' \psi. \varphi' \preceq \Phi \longrightarrow r \ \varphi' \ \psi \longrightarrow \text{all-subformula-st test-symb } \varphi' \longrightarrow \text{all-subformula-st test-symb } \psi$  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:  $\forall c \ \xi \ \varphi \ \xi' \ \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \ \varphi \ \varphi' \longrightarrow \text{wf-conn } c \ (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c \ (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$

### Invariant while lifting of the rewriting relation

The condition  $\varphi \preceq \Phi$  (that will be used with  $\Phi = \varphi$  most of the time) is here to ensure that the recursive conditions on  $\Phi$  will moreover hold for the subterm we are rewriting. For example if

there is no equivalence symbol in  $\Phi$ , we do not have to care about equivalence symbols in the two previous assumptions.

**lemma** *propo-rew-step-inv-stay'*:

**fixes**  $r:: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  **and**  $\text{test-symb}:: 'v \text{ propo} \Rightarrow \text{bool}$  **and**  $x:: 'v$   
**and**  $\varphi \psi \Phi:: 'v \text{ propo}$   
**assumes**  $H: \forall \varphi' \psi. \varphi' \preceq \Phi \longrightarrow r \varphi' \psi \longrightarrow \text{all-subformula-st test-symb } \varphi'$   
 $\longrightarrow \text{all-subformula-st test-symb } \psi$   
**and**  $H': \forall (c:: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \varphi \varphi'$   
 $\longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi'$   
 $\longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$  **and**  
 $\text{propo-rew-step } r \varphi \psi$  **and**  
 $\varphi \preceq \Phi$  **and**  
 $\text{all-subformula-st test-symb } \varphi$   
**shows**  $\text{all-subformula-st test-symb } \psi$   
 $\langle \text{proof} \rangle$

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**  $r:: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  **and**  $\text{test-symb}:: 'v \text{ propo} \Rightarrow \text{bool}$  **and**  $x:: 'v$   
**and**  $\varphi \psi:: 'v \text{ propo}$   
**assumes**  
 $H: \forall \varphi' \psi. r \varphi' \psi \longrightarrow \text{all-subformula-st test-symb } \varphi' \longrightarrow \text{all-subformula-st test-symb } \psi$  **and**  
 $H': \forall (c:: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi'))$   
 $\longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$  **and**  
 $\text{propo-rew-step } r \varphi \psi$  **and**  
 $\text{all-subformula-st test-symb } \varphi$   
**shows**  $\text{all-subformula-st test-symb } \psi$   
 $\langle \text{proof} \rangle$

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

## Invariant after all rewriting

**lemma** *full-propo-rew-step-inv-stay-with-inc*:

**fixes**  $r:: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  **and**  $\text{test-symb}:: 'v \text{ propo} \Rightarrow \text{bool}$  **and**  $x:: 'v$   
**and**  $\varphi \psi:: 'v \text{ propo}$   
**assumes**  
 $H: \forall \varphi \psi. \text{propo-rew-step } r \varphi \psi \longrightarrow \text{all-subformula-st test-symb } \varphi$   
 $\longrightarrow \text{all-subformula-st test-symb } \psi$  **and**  
 $H': \forall (c:: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \varphi \varphi'$   
 $\longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi'$   
 $\longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$  **and**  
 $\varphi \preceq \Phi$  **and**  
 $\text{full: full } (\text{propo-rew-step } r) \varphi \psi$  **and**  
 $\text{init: all-subformula-st test-symb } \varphi$   
**shows**  $\text{all-subformula-st test-symb } \psi$   
 $\langle \text{proof} \rangle$

**lemma** *full-propo-rew-step-inv-stay'*:

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

$\longrightarrow \text{test-symb } (\text{conn } c \ (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$  **and**  
*full*: *full* (*propo-rew-step* *r*)  $\varphi \ \psi$  **and**  
*init*: *all-subformula-st test-symb*  $\varphi$   
**shows** *all-subformula-st test-symb*  $\psi$   
 <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**  $\varphi \ \psi$  :: '*v propo*  
**assumes**  
*H*:  $\forall \varphi \ \psi. \ r \ \varphi \ \psi \longrightarrow \text{all-subformula-st test-symb } \varphi \longrightarrow \text{all-subformula-st test-symb } \psi$  **and**  
*H'*:  $\forall (c:: 'v \text{ connective}) \ \xi \ \varphi \ \xi' \ \varphi'. \ \text{wf-conn } c \ (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c \ (\xi @ \varphi \# \xi'))$   
 $\longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$  **and**  
*full*: *full* (*propo-rew-step* *r*)  $\varphi \ \psi$  **and**  
*init*: *all-subformula-st test-symb*  $\varphi$   
**shows** *all-subformula-st test-symb*  $\psi$   
 <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**  $\varphi \ \psi$  :: '*v propo*  
**assumes**  
*H*:  $\forall \varphi \ \psi. \ r \ \varphi \ \psi \longrightarrow \text{all-subformula-st test-symb } \varphi \longrightarrow \text{all-subformula-st test-symb } \psi$  **and**  
*H'*:  $\forall (c:: 'v \text{ connective}) \ l \ l'. \ \text{wf-conn } c \ l \longrightarrow \text{wf-conn } c \ l'$   
 $\longrightarrow (\text{test-symb } (\text{conn } c \ l) \longleftrightarrow \text{test-symb } (\text{conn } c \ l'))$  **and**  
*full*: *full* (*propo-rew-step* *r*)  $\varphi \ \psi$  **and**  
*init*: *all-subformula-st test-symb*  $\varphi$   
**shows** *all-subformula-st test-symb*  $\psi$   
 <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 separately.

### 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*  $\varphi \ \psi$ ) (*FAnd* (*FImp*  $\varphi \ \psi$ ) (*FImp*  $\psi \ \varphi$ ))

**lemma** *elim-equiv-transformation-consistent*:

$A \models \text{FEq } \varphi \ \psi \longleftrightarrow A \models \text{FAnd } (\text{FImp } \varphi \ \psi) (\text{FImp } \psi \ \varphi)$

$\langle \text{proof} \rangle$

**lemma** *elim-equiv-explicit*:  $\text{elim-equiv } \varphi \ \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$   
 $\langle \text{proof} \rangle$

**lemma** *elim-equiv-consistent*: *preserves-un-sat elim-equiv*  
 $\langle \text{proof} \rangle$

**lemma** *elimEquiv-lifted-consistant*:  
*preserves-un-sat (full (propo-rew-step elim-equiv))*  
 $\langle \text{proof} \rangle$

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

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

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

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

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

**lemma** *no-equiv-eq[simp]*:  
**fixes**  $\varphi \ \psi :: 'v \text{ propo}$   
**shows**  
 $\neg \text{no-equiv } (\text{FEq } \varphi \ \psi)$   
 $\text{no-equiv } FT$   
 $\text{no-equiv } FF$   
 $\langle \text{proof} \rangle$

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**  $\varphi \ \psi :: 'v \text{ propo}$   
**shows**  
 $\text{no-equiv } (\text{FNot } \varphi) \longleftrightarrow \text{no-equiv } \varphi$   
 $\text{no-equiv } (\text{FAnd } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$   
 $\text{no-equiv } (\text{FOr } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$   
 $\text{no-equiv } (\text{FImp } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$   
 $\langle \text{proof} \rangle$

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**  $\varphi :: 'v \text{ propo}$   
**assumes** *no-equiv*:  $\neg \text{no-equiv } \varphi$   
**shows**  $\exists \psi \ \psi'. \psi \preceq \varphi \wedge \text{elim-equiv } \psi \ \psi'$   
 $\langle \text{proof} \rangle$

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*:  
 $\text{full } (\text{propo-rew-step elim-equiv}) \varphi \psi \implies \text{no-equiv } \psi$   
 $\langle \text{proof} \rangle$

### 3.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

**inductive** *elim-imp* :: 'v propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool **where**  
 $[\text{simp}]: \text{elim-imp } (F\text{Imp } \varphi \psi) (F\text{Or } (F\text{Not } \varphi) \psi)$

**lemma** *elim-imp-transformation-consistent*:  
 $A \models F\text{Imp } \varphi \psi \longleftrightarrow A \models F\text{Or } (F\text{Not } \varphi) \psi$   
 $\langle \text{proof} \rangle$

**lemma** *elim-imp-explicit*:  $\text{elim-imp } \varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$   
 $\langle \text{proof} \rangle$

**lemma** *elim-imp-consistent*: *preserves-un-sat elim-imp*  
 $\langle \text{proof} \rangle$

**lemma** *elim-imp-lifted-consistant*:  
 $\text{preserves-un-sat } (\text{full } (\text{propo-rew-step elim-imp}))$   
 $\langle \text{proof} \rangle$

**fun** *no-imp-symb* **where**  
 $\text{no-imp-symb } (F\text{Imp } -) = \text{False} \mid$   
 $\text{no-imp-symb } - = \text{True}$

**lemma** *no-imp-symb-conn-characterization*:  
 $\text{wf-conn } c \ l \implies \text{no-imp-symb } (\text{conn } c \ l) \longleftrightarrow c \neq C\text{Imp}$   
 $\langle \text{proof} \rangle$

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

**lemma** *no-imp-Imp[simp]*:  
 $\neg \text{no-imp } (F\text{Imp } \varphi \psi)$   
 $\text{no-imp } FT$   
 $\text{no-imp } FF$   
 $\langle \text{proof} \rangle$

**lemma** *all-subformula-st-decomp-explicit-imp[simp]*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**shows**  
 $\text{no-imp } (F\text{Not } \varphi) \longleftrightarrow \text{no-imp } \varphi$   
 $\text{no-imp } (F\text{And } \varphi \psi) \longleftrightarrow (\text{no-imp } \varphi \wedge \text{no-imp } \psi)$   
 $\text{no-imp } (F\text{Or } \varphi \psi) \longleftrightarrow (\text{no-imp } \varphi \vee \text{no-imp } \psi)$   
 $\langle \text{proof} \rangle$

Invariant of the *elim-imp* transformation

**lemma** *elim-imp-no-equiv*:  
 $\text{elim-imp } \varphi \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$

$\langle \text{proof} \rangle$

**lemma** *elim-imp-inv*:

**fixes**  $\varphi \ \psi :: 'v \ \text{propo}$

**assumes** *full* (*propo-rew-step elim-imp*)  $\varphi \ \psi$  **and** *no-equiv*  $\varphi$

**shows** *no-equiv*  $\psi$

$\langle \text{proof} \rangle$

**lemma** *no-no-imp-elim-imp-step-exists*:

**fixes**  $\varphi :: 'v \ \text{propo}$

**assumes** *no-equiv*:  $\neg \text{no-imp } \varphi$

**shows**  $\exists \psi \ \psi'. \ \psi \preceq \varphi \wedge \text{elim-imp } \psi \ \psi'$

$\langle \text{proof} \rangle$

**lemma** *no-imp-full-propo-rew-step-elim-imp*: *full* (*propo-rew-step elim-imp*)  $\varphi \ \psi \implies \text{no-imp } \psi$

$\langle \text{proof} \rangle$

### 3.5.3 Eliminate all the True and False in the formula

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

**inductive** *elimTB* **where**

*ElimTB1*: *elimTB* (*FAnd*  $\varphi \ FT$ )  $\varphi \mid$

*ElimTB1'*: *elimTB* (*FAnd*  $FT \ \varphi$ )  $\varphi \mid$

*ElimTB2*: *elimTB* (*FAnd*  $\varphi \ FF$ )  $FF \mid$

*ElimTB2'*: *elimTB* (*FAnd*  $FF \ \varphi$ )  $FF \mid$

*ElimTB3*: *elimTB* (*FOr*  $\varphi \ FT$ )  $FT \mid$

*ElimTB3'*: *elimTB* (*FOr*  $FT \ \varphi$ )  $FT \mid$

*ElimTB4*: *elimTB* (*FOr*  $\varphi \ FF$ )  $\varphi \mid$

*ElimTB4'*: *elimTB* (*FOr*  $FF \ \varphi$ )  $\varphi \mid$

*ElimTB5*: *elimTB* (*FNot*  $FT$ )  $FF \mid$

*ElimTB6*: *elimTB* (*FNot*  $FF$ )  $FT$

**lemma** *elimTB-consistent*: *preserves-un-sat elimTB*

$\langle \text{proof} \rangle$

**inductive** *no-T-F-symb* ::  $'v \ \text{propo} \Rightarrow \text{bool}$  **where**

*no-T-F-symb-comp*:  $c \neq CF \implies c \neq CT \implies \text{wf-conn } c \ l \implies (\forall \varphi \in \text{set } l. \ \varphi \neq FT \wedge \varphi \neq FF) \implies \text{no-T-F-symb } (\text{conn } c \ l)$

**lemma** *wf-conn-no-T-F-symb-iff[simp]*:

*wf-conn*  $c \ \psi s \implies$

*no-T-F-symb* (*conn*  $c \ \psi s$ )  $\longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in \text{set } \psi s. \ \psi \neq FF \wedge \psi \neq FT))$

$\langle \text{proof} \rangle$

**lemma** *wf-conn-no-T-F-symb-iff-explicit[simp]*:

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

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

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

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

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

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

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

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

Actually it is not possible to remove 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*  $\varphi \implies \text{no-T-F-symb-except-toplevel } \varphi$

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

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

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

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

**lemma** *no-T-F-symb-except-top-level-false-example[simp]*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes**  $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$   
**shows**  
 $\neg \text{no-T-F-symb-except-toplevel } (FAnd \ \varphi \ \psi)$   
 $\neg \text{no-T-F-symb-except-toplevel } (FOr \ \varphi \ \psi)$   
 $\neg \text{no-T-F-symb-except-toplevel } (FImp \ \varphi \ \psi)$   
 $\neg \text{no-T-F-symb-except-toplevel } (FEq \ \varphi \ \psi)$   
 $\langle \text{proof} \rangle$

**lemma** *no-T-F-symb-except-top-level-false-not[simp]*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes**  $\varphi = FT \vee \varphi = FF$   
**shows**  
 $\neg \text{no-T-F-symb-except-toplevel } (FNot \ \varphi)$   
 $\langle \text{proof} \rangle$

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

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

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**  
 $\text{no-T-F} \equiv \text{all-subformula-st no-T-F-symb}$

**lemma** *no-T-F-except-top-level-false*:  
**fixes**  $l :: 'v \text{ propo list}$  **and**  $c :: 'v \text{ connective}$   
**assumes**  $\text{wf-conn } c \ l$   
**and**  $FT \in \text{set } l \vee FF \in \text{set } l$   
**shows**  $\neg \text{no-T-F-except-top-level } (\text{conn } c \ l)$   
 $\langle \text{proof} \rangle$

**lemma** *no-T-F-except-top-level-false-example[simp]*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes**  $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$   
**shows**  
 $\neg \text{no-T-F-except-top-level } (FAnd \ \varphi \ \psi)$   
 $\neg \text{no-T-F-except-top-level } (FOr \ \varphi \ \psi)$   
 $\neg \text{no-T-F-except-top-level } (FEq \ \varphi \ \psi)$   
 $\neg \text{no-T-F-except-top-level } (FImp \ \varphi \ \psi)$   
 $\langle \text{proof} \rangle$

**lemma** *no-T-F-symb-except-toplevel-no-T-F-symb*:  
 $\text{no-T-F-symb-except-toplevel } \varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies \text{no-T-F-symb } \varphi$   
 $\langle \text{proof} \rangle$

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*:  
 $\text{no-T-F-except-top-level } \varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies \text{no-T-F } \varphi$   
 $\langle \text{proof} \rangle$

**lemma** *no-T-F-no-T-F-except-top-level*:  
 $\text{no-T-F } \varphi \implies \text{no-T-F-except-top-level } \varphi$



$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

**lemma** *elimTB-inv*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes** *full (propo-rew-step elimTB)  $\varphi \psi$*   
**and** *no-equiv  $\varphi$  and no-imp  $\varphi$*   
**shows** *no-equiv  $\psi$  and no-imp  $\psi$*

$\langle \text{proof} \rangle$

**lemma** *elimTB-full-propo-rew-step*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes** *no-equiv  $\varphi$  and no-imp  $\varphi$  and full (propo-rew-step elimTB)  $\varphi \psi$*   
**shows** *no-T-F-except-top-level  $\psi$*

$\langle \text{proof} \rangle$

### 3.5.4 PushNeg

Push the negation inside the formula, until the literal.

**inductive** *pushNeg* **where**

*PushNeg1[simp]: pushNeg (FNot (FAnd  $\varphi \psi$ )) (FOr (FNot  $\varphi$ ) (FNot  $\psi$ )) |*  
*PushNeg2[simp]: pushNeg (FNot (FOr  $\varphi \psi$ )) (FAnd (FNot  $\varphi$ ) (FNot  $\psi$ )) |*  
*PushNeg3[simp]: pushNeg (FNot (FNot  $\varphi$ ))  $\varphi$*

**lemma** *pushNeg-transformation-consistent*:

$A \models \text{FNot (FAnd } \varphi \psi) \longleftrightarrow A \models (\text{FOr (FNot } \varphi) (\text{FNot } \psi))$   
 $A \models \text{FNot (FOr } \varphi \psi) \longleftrightarrow A \models (\text{FAnd (FNot } \varphi) (\text{FNot } \psi))$   
 $A \models \text{FNot (FNot } \varphi) \longleftrightarrow A \models \varphi$

$\langle \text{proof} \rangle$

**lemma** *pushNeg-explicit*:  $\text{pushNeg } \varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$

$\langle \text{proof} \rangle$

**lemma** *pushNeg-consistent*: *preserves-un-sat pushNeg*

$\langle \text{proof} \rangle$

**lemma** *pushNeg-lifted-consistant*:

*preserves-un-sat (full (propo-rew-step pushNeg))*

$\langle \text{proof} \rangle$

**fun** *simple* **where**

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

**lemma** *simple-decomp*:

*simple  $\varphi \longleftrightarrow (\varphi = \text{FT} \vee \varphi = \text{FF} \vee (\exists x. \varphi = \text{FVar } x))$*

$\langle \text{proof} \rangle$

**lemma** *subformula-conn-decomp-simple*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes** *s: simple  $\psi$*   
**shows**  $\varphi \preceq \text{FNot } \psi \longleftrightarrow (\varphi = \text{FNot } \psi \vee \varphi = \psi)$

$\langle \text{proof} \rangle$

**lemma** *subformula-conn-decomp-explicit*[simp]:

**fixes**  $\varphi :: 'v \text{ propo}$  **and**  $x :: 'v$

**shows**

$\varphi \preceq \text{FNot } FT \longleftrightarrow (\varphi = \text{FNot } FT \vee \varphi = FT)$

$\varphi \preceq \text{FNot } FF \longleftrightarrow (\varphi = \text{FNot } FF \vee \varphi = FF)$

$\varphi \preceq \text{FNot } (\text{FVar } x) \longleftrightarrow (\varphi = \text{FNot } (\text{FVar } x) \vee \varphi = \text{FVar } x)$

$\langle \text{proof} \rangle$

**fun** *simple-not-symb* **where**

*simple-not-symb* ( $\text{FNot } \varphi$ ) = (*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]:

$\neg \text{simple-not } (\text{FNot } (\text{FAnd } \varphi \ \psi))$

$\neg \text{simple-not } (\text{FNot } (\text{FOr } \varphi \ \psi))$

$\langle \text{proof} \rangle$

**lemma** *simple-not-step-exists*:

**fixes**  $\varphi \ \psi :: 'v \text{ propo}$

**assumes** *no-equiv*  $\varphi$  **and** *no-imp*  $\varphi$

**shows**  $\psi \preceq \varphi \implies \neg \text{simple-not-symb } \psi \implies \exists \psi'. \text{pushNeg } \psi \ \psi'$

$\langle \text{proof} \rangle$

**lemma** *simple-not-rew*:

**fixes**  $\varphi :: 'v \text{ propo}$

**assumes** *noTB*:  $\neg \text{simple-not } \varphi$  **and** *no-equiv*: *no-equiv*  $\varphi$  **and** *no-imp*: *no-imp*  $\varphi$

**shows**  $\exists \psi \ \psi'. \psi \preceq \varphi \wedge \text{pushNeg } \psi \ \psi'$

$\langle \text{proof} \rangle$

**lemma** *no-T-F-except-top-level-pushNeg1*:

*no-T-F-except-top-level* ( $\text{FNot } (\text{FAnd } \varphi \ \psi)$ )  $\implies$  *no-T-F-except-top-level* ( $\text{FOr } (\text{FNot } \varphi) (\text{FNot } \psi)$ )

$\langle \text{proof} \rangle$

**lemma** *no-T-F-except-top-level-pushNeg2*:

*no-T-F-except-top-level* ( $\text{FNot } (\text{FOr } \varphi \ \psi)$ )  $\implies$  *no-T-F-except-top-level* ( $\text{FAnd } (\text{FNot } \varphi) (\text{FNot } \psi)$ )

$\langle \text{proof} \rangle$

**lemma** *no-T-F-symb-pushNeg*:

*no-T-F-symb* ( $\text{FOr } (\text{FNot } \varphi') (\text{FNot } \psi')$ )

*no-T-F-symb* ( $\text{FAnd } (\text{FNot } \varphi') (\text{FNot } \psi')$ )

*no-T-F-symb* ( $\text{FNot } (\text{FNot } \varphi')$ )

$\langle \text{proof} \rangle$

**lemma** *propo-rew-step-pushNeg-no-T-F-symb*:

*propo-rew-step pushNeg*  $\varphi \ \psi \implies$  *no-T-F-except-top-level*  $\varphi \implies$  *no-T-F-symb*  $\varphi \implies$  *no-T-F-symb*  $\psi$

$\langle \text{proof} \rangle$

**lemma** *propo-rew-step-pushNeg-no-T-F*:

*propo-rew-step pushNeg*  $\varphi \ \psi \implies$  *no-T-F*  $\varphi \implies$  *no-T-F*  $\psi$

$\langle \text{proof} \rangle$

**lemma** *pushNeg-inv*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$

**assumes** *full* (*propo-rew-step* *pushNeg*)  $\varphi \psi$

**and** *no-equiv*  $\varphi$  **and** *no-imp*  $\varphi$  **and** *no-T-F-except-top-level*  $\varphi$

**shows** *no-equiv*  $\psi$  **and** *no-imp*  $\psi$  **and** *no-T-F-except-top-level*  $\psi$

$\langle \text{proof} \rangle$

**lemma** *pushNeg-full-propo-rew-step*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$

**assumes**

*no-equiv*  $\varphi$  **and**

*no-imp*  $\varphi$  **and**

*full* (*propo-rew-step* *pushNeg*)  $\varphi \psi$  **and**

*no-T-F-except-top-level*  $\varphi$

**shows** *simple-not*  $\psi$

$\langle \text{proof} \rangle$

### 3.5.5 Push inside

**inductive** *push-conn-inside* ::  $'v \text{ connective} \Rightarrow 'v \text{ connective} \Rightarrow 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$

**for**  $c c' :: 'v \text{ connective}$  **where**

*push-conn-inside-l[simp]*:  $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$

$\Longrightarrow \text{push-conn-inside } c c' (\text{conn } c [\text{conn } c' [\varphi 1, \varphi 2], \psi])$

$(\text{conn } c' [\text{conn } c [\varphi 1, \psi], \text{conn } c [\varphi 2, \psi]]) \mid$

*push-conn-inside-r[simp]*:  $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$

$\Longrightarrow \text{push-conn-inside } c c' (\text{conn } c [\psi, \text{conn } c' [\varphi 1, \varphi 2]])$

$(\text{conn } c' [\text{conn } c [\psi, \varphi 1], \text{conn } c [\psi, \varphi 2]])$

**lemma** *push-conn-inside-explicit*:  $\text{push-conn-inside } c c' \varphi \psi \Longrightarrow \forall A. A \models \varphi \longleftrightarrow A \models \psi$

$\langle \text{proof} \rangle$

**lemma** *push-conn-inside-consistent*: *preserves-un-sat* (*push-conn-inside*  $c c'$ )

$\langle \text{proof} \rangle$

**lemma** *propo-rew-step-push-conn-inside[simp]*:

$\neg \text{propo-rew-step } (\text{push-conn-inside } c c') FT \psi \neg \text{propo-rew-step } (\text{push-conn-inside } c c') FF \psi$

$\langle \text{proof} \rangle$

**inductive** *not-c-in-c'-symb*::  $'v \text{ connective} \Rightarrow 'v \text{ connective} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  **for**  $c c'$  **where**

*not-c-in-c'-symb-l[simp]*:  $\text{wf-conn } c [\text{conn } c' [\varphi, \varphi'], \psi] \Longrightarrow \text{wf-conn } c' [\varphi, \varphi']$

$\Longrightarrow \text{not-c-in-c'-symb } c c' (\text{conn } c [\text{conn } c' [\varphi, \varphi'], \psi]) \mid$

*not-c-in-c'-symb-r[simp]*:  $\text{wf-conn } c [\psi, \text{conn } c' [\varphi, \varphi']] \Longrightarrow \text{wf-conn } c' [\varphi, \varphi']$

$\Longrightarrow \text{not-c-in-c'-symb } c c' (\text{conn } c [\psi, \text{conn } c' [\varphi, \varphi']])$

**abbreviation** *c-in-c'-symb*  $c c' \varphi \equiv \neg \text{not-c-in-c'-symb } c c' \varphi$

**lemma** *c-in-c'-symb-simp*:

$\text{not-c-in-c'-symb } c c' \xi \Longrightarrow \xi = FF \vee \xi = FT \vee \xi = FVar x \vee \xi = FNot FF \vee \xi = FNot FT$

$\vee \xi = FNot (FVar x) \Longrightarrow \text{False}$

$\langle \text{proof} \rangle$

**lemma** *c-in-c'-symb-simp'[simp]:*

$\neg \text{not-c-in-c'-symb } c \ c' \text{ } FF$   
 $\neg \text{not-c-in-c'-symb } c \ c' \text{ } FT$   
 $\neg \text{not-c-in-c'-symb } c \ c' \text{ } (FVar \ x)$   
 $\neg \text{not-c-in-c'-symb } c \ c' \text{ } (FNot \ FF)$   
 $\neg \text{not-c-in-c'-symb } c \ c' \text{ } (FNot \ FT)$   
 $\neg \text{not-c-in-c'-symb } c \ c' \text{ } (FNot \ (FVar \ x))$   
 $\langle \text{proof} \rangle$

**definition** *c-in-c'-only where*

*c-in-c'-only*  $c \ c' \equiv \text{all-subformula-st } (c\text{-in-c'-symb } c \ c')$

**lemma** *c-in-c'-only-simp[simp]:*

*c-in-c'-only*  $c \ c' \text{ } FF$   
*c-in-c'-only*  $c \ c' \text{ } FT$   
*c-in-c'-only*  $c \ c' \text{ } (FVar \ x)$   
*c-in-c'-only*  $c \ c' \text{ } (FNot \ FF)$   
*c-in-c'-only*  $c \ c' \text{ } (FNot \ FT)$   
*c-in-c'-only*  $c \ c' \text{ } (FNot \ (FVar \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *not-c-in-c'-symb-commute:*

$\text{not-c-in-c'-symb } c \ c' \ \xi \implies \text{wf-conn } c \ [\varphi, \psi] \implies \xi = \text{conn } c \ [\varphi, \psi]$   
 $\implies \text{not-c-in-c'-symb } c \ c' \text{ } (\text{conn } c \ [\psi, \varphi])$

$\langle \text{proof} \rangle$

**lemma** *not-c-in-c'-symb-commute':*

$\text{wf-conn } c \ [\varphi, \psi] \implies c\text{-in-c'-symb } c \ c' \text{ } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow c\text{-in-c'-symb } c \ c' \text{ } (\text{conn } c \ [\psi, \varphi])$   
 $\langle \text{proof} \rangle$

**lemma** *not-c-in-c'-comm:*

**assumes** *wf*:  $\text{wf-conn } c \ [\varphi, \psi]$   
**shows**  $c\text{-in-c'-only } c \ c' \text{ } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow c\text{-in-c'-only } c \ c' \text{ } (\text{conn } c \ [\psi, \varphi])$  (**is**  $?A \longleftrightarrow ?B$ )  
 $\langle \text{proof} \rangle$

**lemma** *not-c-in-c'-simp[simp]:*

**fixes**  $\varphi1 \ \varphi2 \ \psi :: 'v \text{ propo}$  **and**  $x :: 'v$   
**shows**  
 $c\text{-in-c'-symb } c \ c' \text{ } FT$   
 $c\text{-in-c'-symb } c \ c' \text{ } FF$   
 $c\text{-in-c'-symb } c \ c' \text{ } (FVar \ x)$   
 $\text{wf-conn } c \ [\text{conn } c' \ [\varphi1, \varphi2], \psi] \implies \text{wf-conn } c' \ [\varphi1, \varphi2]$   
 $\implies \neg c\text{-in-c'-only } c \ c' \text{ } (\text{conn } c \ [\text{conn } c' \ [\varphi1, \varphi2], \psi])$   
 $\langle \text{proof} \rangle$

**lemma** *c-in-c'-symb-not[simp]:*

**fixes**  $c \ c' :: 'v \text{ connective}$  **and**  $\psi :: 'v \text{ propo}$   
**shows**  $c\text{-in-c'-symb } c \ c' \text{ } (FNot \ \psi)$   
 $\langle \text{proof} \rangle$

**lemma** *c-in-c'-symb-step-exists:*

**fixes**  $\varphi :: 'v \text{ propo}$   
**assumes**  $c: c = CAnd \vee c = COr$  **and**  $c': c' = CAnd \vee c' = COr$

**shows**  $\psi \preceq \varphi \implies \neg \text{c-in-c'-symb } c \ c' \ \psi \implies \exists \psi'. \text{push-conn-inside } c \ c' \ \psi \ \psi'$   
 <proof>

**lemma** *c-in-c'-symb-rew*:

**fixes**  $\varphi :: 'v \text{ propo}$

**assumes** *noTB*:  $\neg \text{c-in-c'-only } c \ c' \ \varphi$

**and**  $c: c = CAnd \vee c = COr$  **and**  $c': c' = CAnd \vee c' = COr$

**shows**  $\exists \psi \ \psi'. \psi \preceq \varphi \wedge \text{push-conn-inside } c \ c' \ \psi \ \psi'$

<proof>

**lemma** *push-conn-insidec-in-c'-symb-no-T-F*:

**fixes**  $\varphi \ \psi :: 'v \text{ propo}$

**shows** *propo-rew-step* (*push-conn-inside*  $c \ c'$ )  $\varphi \ \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$

<proof>

**lemma** *simple-propo-rew-step-push-conn-inside-inv*:

*propo-rew-step* (*push-conn-inside*  $c \ c'$ )  $\varphi \ \psi \implies \text{simple } \varphi \implies \text{simple } \psi$

<proof>

**lemma** *simple-propo-rew-step-inv-push-conn-inside-simple-not*:

**fixes**  $c \ c' :: 'v \text{ connective}$  **and**  $\varphi \ \psi :: 'v \text{ propo}$

**shows** *propo-rew-step* (*push-conn-inside*  $c \ c'$ )  $\varphi \ \psi \implies \text{simple-not } \varphi \implies \text{simple-not } \psi$

<proof>

**lemma** *propo-rew-step-push-conn-inside-simple-not*:

**fixes**  $\varphi \ \varphi' :: 'v \text{ propo}$  **and**  $\xi \ \xi' :: 'v \text{ propo list}$  **and**  $c :: 'v \text{ connective}$

**assumes**

*propo-rew-step* (*push-conn-inside*  $c \ c'$ )  $\varphi \ \varphi'$  **and**

*wf-conn*  $c \ (\xi @ \varphi \# \xi')$  **and**

*simple-not-symb* (*conn*  $c \ (\xi @ \varphi \# \xi')$ ) **and**

*simple-not-symb*  $\varphi'$

**shows** *simple-not-symb* (*conn*  $c \ (\xi @ \varphi' \# \xi')$ )

<proof>

**lemma** *push-conn-inside-not-true-false*:

*push-conn-inside*  $c \ c' \ \varphi \ \psi \implies \psi \neq FT \wedge \psi \neq FF$

<proof>

**lemma** *push-conn-inside-inv*:

**fixes**  $\varphi \ \psi :: 'v \text{ propo}$

**assumes** *full* (*propo-rew-step* (*push-conn-inside*  $c \ c'$ ))  $\varphi \ \psi$

**and** *no-equiv*  $\varphi$  **and** *no-imp*  $\varphi$  **and** *no-T-F-except-top-level*  $\varphi$  **and** *simple-not*  $\varphi$

**shows** *no-equiv*  $\psi$  **and** *no-imp*  $\psi$  **and** *no-T-F-except-top-level*  $\psi$  **and** *simple-not*  $\psi$

<proof>

**lemma** *push-conn-inside-full-propo-rew-step*:

**fixes**  $\varphi \ \psi :: 'v \text{ propo}$

**assumes**

*no-equiv*  $\varphi$  **and**

*no-imp*  $\varphi$  **and**

*full* (*propo-rew-step* (*push-conn-inside*  $c \ c'$ ))  $\varphi \ \psi$  **and**

*no-T-F-except-top-level*  $\varphi$  **and**

*simple-not*  $\varphi$  **and**  
 $c = CAnd \vee c = COr$  **and**  
 $c' = CAnd \vee c' = COr$   
**shows** *c-in-c'-only*  $c$   $c'$   $\psi$   
 $\langle proof \rangle$

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

**inductive** *only-c-inside-symb* :: '*v* connective  $\Rightarrow$  '*v* propo  $\Rightarrow$  bool **for**  $c ::$  '*v* connective **where**  
*simple-only-c-inside*[*simp*]: *simple*  $\varphi \Rightarrow$  *only-c-inside-symb*  $c$   $\varphi$  |  
*simple-cnot-only-c-inside*[*simp*]: *simple*  $\varphi \Rightarrow$  *only-c-inside-symb*  $c$  (*FNot*  $\varphi$ ) |  
*only-c-inside-into-only-c-inside*: *wf-conn*  $c$   $l \Rightarrow$  *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*  $x$ )  $\langle proof \rangle$

**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*  $\psi$   
 $\vee (\exists \varphi'. \psi = FNot \varphi' \wedge \text{simple } \varphi')$   
 $\vee (\exists l. \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l)$ )  
 $\langle proof \rangle$

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

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

**lemma** *only-c-inside-decomp*:  
*only-c-inside*  $c$   $\varphi \longleftrightarrow$   
 $(\forall \psi. \psi \preceq \varphi \longrightarrow (\text{simple } \psi \vee (\exists \varphi'. \psi = FNot \varphi' \wedge \text{simple } \varphi') \vee (\exists l. \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l)))$   
 $\langle proof \rangle$

**lemma** *only-c-inside-c-c'-false*:  
**fixes**  $c \ c' ::$  '*v* connective **and**  $l ::$  '*v* propo list **and**  $\varphi ::$  '*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*  
 $\langle proof \rangle$

**lemma** *only-c-inside-implies-c-in-c'-symb*:  
**assumes**  $\delta: c \neq c'$  **and**  $c: c = CAnd \vee c = COr$  **and**  $c': c' = CAnd \vee c' = COr$   
**shows** *only-c-inside*  $c$   $\varphi \Rightarrow$  *c-in-c'-symb*  $c$   $c' \ \varphi$   
 $\langle proof \rangle$

**lemma** *c-in-c'-symb-decomp-level1*:  
**fixes**  $l :: 'v \text{ propo list}$  **and**  $c \ c' \text{ ca} :: 'v \text{ connective}$   
**shows**  $\text{wf-conn } ca \ l \implies ca \neq c \implies c\text{-in-}c'\text{-symb } c \ c' (\text{conn } ca \ l)$   
 $\langle \text{proof} \rangle$

**lemma** *only-c-inside-implies-c-in-c'-only*:  
**assumes**  $\delta: c \neq c' \text{ and } c: c = CAnd \vee c = COr \text{ and } c': c' = CAnd \vee c' = COr$   
**shows**  $\text{only-c-inside } c \ \varphi \implies c\text{-in-}c'\text{-only } c \ c' \ \varphi$   
 $\langle \text{proof} \rangle$

**lemma** *c-in-c'-symb-c-implies-only-c-inside*:  
**assumes**  $\delta: c = CAnd \vee c = COr \ c' = CAnd \vee c' = COr \ c \neq c' \text{ and } \text{wf}: \text{wf-conn } c \ [\varphi, \psi]$   
**and** *inv*:  $\text{no-equiv } (\text{conn } c \ l) \ \text{no-imp } (\text{conn } c \ l) \ \text{simple-not } (\text{conn } c \ l)$   
**shows**  $\text{wf-conn } c \ l \implies c\text{-in-}c'\text{-only } c \ c' (\text{conn } c \ l) \implies (\forall \psi \in \text{set } l. \text{only-c-inside } c \ \psi)$   
 $\langle \text{proof} \rangle$

## Push Conjunction

**definition** *pushConj* **where**  $\text{pushConj} = \text{push-conn-inside } CAnd \ COr$

**lemma** *pushConj-consistent*: *preserves-un-sat pushConj*  
 $\langle \text{proof} \rangle$

**definition** *and-in-or-symb* **where**  $\text{and-in-or-symb} = c\text{-in-}c'\text{-symb } CAnd \ COr$

**definition** *and-in-or-only* **where**  
 $\text{and-in-or-only} = \text{all-subformula-st } (c\text{-in-}c'\text{-symb } CAnd \ COr)$

**lemma** *pushConj-inv*:  
**fixes**  $\varphi \ \psi :: 'v \text{ propo}$   
**assumes** *full*  $(\text{propo-rew-step } \text{pushConj}) \ \varphi \ \psi$   
**and** *no-equiv*  $\varphi$  **and** *no-imp*  $\varphi$  **and** *no-T-F-except-top-level*  $\varphi$  **and** *simple-not*  $\varphi$   
**shows** *no-equiv*  $\psi$  **and** *no-imp*  $\psi$  **and** *no-T-F-except-top-level*  $\psi$  **and** *simple-not*  $\psi$   
 $\langle \text{proof} \rangle$

**lemma** *pushConj-full-propo-rew-step*:  
**fixes**  $\varphi \ \psi :: 'v \text{ propo}$   
**assumes**  
*no-equiv*  $\varphi$  **and**  
*no-imp*  $\varphi$  **and**  
*full*  $(\text{propo-rew-step } \text{pushConj}) \ \varphi \ \psi$  **and**  
*no-T-F-except-top-level*  $\varphi$  **and**  
*simple-not*  $\varphi$   
**shows** *and-in-or-only*  $\psi$   
 $\langle \text{proof} \rangle$

## Push Disjunction

**definition** *pushDisj* **where**  $\text{pushDisj} = \text{push-conn-inside } COr \ CAnd$

**lemma** *pushDisj-consistent*: *preserves-un-sat pushDisj*  
 $\langle \text{proof} \rangle$



**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]*:  
 $\sim \text{or-in-and-only } (FOr \ (FAnd \ \psi1 \ \psi2) \ \varphi')$   
 $\langle \text{proof} \rangle$

**lemma** *pushDisj-inv*:

**fixes**  $\varphi \ \psi :: 'v \text{ propo}$

**assumes** *full* (*propo-rew-step* *pushDisj*)  $\varphi \ \psi$

**and** *no-equiv*  $\varphi$  **and** *no-imp*  $\varphi$  **and** *no-T-F-except-top-level*  $\varphi$  **and** *simple-not*  $\varphi$

**shows** *no-equiv*  $\psi$  **and** *no-imp*  $\psi$  **and** *no-T-F-except-top-level*  $\psi$  **and** *simple-not*  $\psi$

$\langle \text{proof} \rangle$

**lemma** *pushDisj-full-propo-rew-step*:

**fixes**  $\varphi \ \psi :: 'v \text{ propo}$

**assumes**

*no-equiv*  $\varphi$  **and**

*no-imp*  $\varphi$  **and**

*full* (*propo-rew-step* *pushDisj*)  $\varphi \ \psi$  **and**

*no-T-F-except-top-level*  $\varphi$  **and**

*simple-not*  $\varphi$

**shows** *or-in-and-only*  $\psi$

$\langle \text{proof} \rangle$

## 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*  $\Rightarrow$  *'a* *propo*  $\Rightarrow$  *bool* **for** *c* **where**

*simple-is-grouped[simp]*: *simple*  $\varphi \Rightarrow$  *grouped-by* *c*  $\varphi$  |

*simple-not-is-grouped[simp]*: *simple*  $\varphi \Rightarrow$  *grouped-by* *c* (*FNot*  $\varphi$ ) |

*connected-is-group[simp]*: *grouped-by* *c*  $\varphi \Rightarrow$  *grouped-by* *c*  $\psi \Rightarrow$  *wf-conn* *c*  $[\varphi, \psi]$   
 $\Rightarrow$  *grouped-by* *c* (*conn* *c*  $[\varphi, \psi]$ )

**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*))

$\langle \text{proof} \rangle$

**lemma** *only-c-inside-symb-c-eq-c'*:

*only-c-inside-symb* *c* (*conn* *c'*  $[\varphi1, \varphi2]$ )  $\Rightarrow$   $c' = CAnd \vee c' = COr \Rightarrow$  *wf-conn* *c'*  $[\varphi1, \varphi2]$

$\Rightarrow c' = c$

$\langle \text{proof} \rangle$

**lemma** *only-c-inside-c-eq-c'*:

*only-c-inside*  $c$  (*conn*  $c'$  [ $\varphi 1$ ,  $\varphi 2$ ])  $\implies c' = CAnd \vee c' = COr \implies wf\text{-}conn\ c' [\varphi 1, \varphi 2] \implies c = c'$   
 $\langle proof \rangle$

**lemma** *only-c-inside-imp-grouped-by*:

**assumes**  $c: c \neq CNot$  **and**  $c': c' = CAnd \vee c' = COr$   
**shows** *only-c-inside*  $c\ \varphi \implies grouped\text{-}by\ c\ \varphi$  (**is**  $?O\ \varphi \implies ?G\ \varphi$ )  
 $\langle proof \rangle$

**lemma** *grouped-by-false*:

*grouped-by*  $c$  (*conn*  $c'$  [ $\varphi$ ,  $\psi$ ])  $\implies c \neq c' \implies wf\text{-}conn\ c' [\varphi, \psi] \implies False$   
 $\langle proof \rangle$

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

**lemma** *simple-cnf*[*simp*]:

*super-grouped-by*  $c\ c'\ FT$   
*super-grouped-by*  $c\ c'\ FF$   
*super-grouped-by*  $c\ c'\ (FVar\ x)$   
*super-grouped-by*  $c\ c'\ (FNot\ FT)$   
*super-grouped-by*  $c\ c'\ (FNot\ FF)$   
*super-grouped-by*  $c\ c'\ (FNot\ (FVar\ x))$   
 $\langle proof \rangle$

**lemma** *c-in-c'-only-super-grouped-by*:

**assumes**  $c: c = CAnd \vee c = COr$  **and**  $c': c' = CAnd \vee c' = COr$  **and**  $cc': c \neq c'$   
**shows** *no-equiv*  $\varphi \implies no\text{-}imp\ \varphi \implies simple\text{-}not\ \varphi \implies c\text{-}in\text{-}c'\text{-}only\ c\ c'\ \varphi$   
 $\implies super\text{-}grouped\text{-}by\ c\ c'\ \varphi$   
(**is**  $?NE\ \varphi \implies ?NI\ \varphi \implies ?SN\ \varphi \implies ?C\ \varphi \implies ?S\ \varphi$ )  
 $\langle proof \rangle$

### 3.6.2 Conjunctive Normal Form

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

**lemma** *or-in-and-only-conjunction-in-disj*:

**shows** *no-equiv*  $\varphi \implies no\text{-}imp\ \varphi \implies simple\text{-}not\ \varphi \implies or\text{-}in\text{-}and\text{-}only\ \varphi \implies is\text{-}conj\text{-}with\text{-}TF\ \varphi$   
 $\langle proof \rangle$

**definition** *is-cnf* **where**

*is-cnf*  $\varphi \equiv is\text{-}conj\text{-}with\text{-}TF\ \varphi \wedge no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\ \varphi$

### Full CNF transformation

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

**definition** *cnf-rew* **where** *cnf-rew* =

(*full* (*propo-rew-step elim-equiv*)) *OO*

$(full\ (propo\text{-}rew\text{-}step\ elim\text{-}imp))\ OO$   
 $(full\ (propo\text{-}rew\text{-}step\ elimTB))\ OO$   
 $(full\ (propo\text{-}rew\text{-}step\ pushNeg))\ OO$   
 $(full\ (propo\text{-}rew\text{-}step\ pushDisj))$

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

**lemma** *cnf-rew-is-cnf: cnf-rew  $\varphi\ \varphi' \implies is\text{-}cnf\ \varphi'$*   
 $\langle proof \rangle$

### 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  $\varphi \implies no\text{-}imp\ \varphi \implies simple\text{-}not\ \varphi \implies and\text{-}in\text{-}or\text{-}only\ \varphi \implies is\text{-}disj\text{-}with\text{-}TF\ \varphi$*   
 $\langle proof \rangle$

**definition** *is-dnf :: 'a propo  $\Rightarrow$  bool where*  
*is-dnf  $\varphi \longleftrightarrow is\text{-}disj\text{-}with\text{-}TF\ \varphi \wedge no\text{-}TF\text{-}except\text{-}top\text{-}level\ \varphi$*

#### Full DNF transform

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

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

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

**theorem** *dnf-transformation-correction:*  
 $dnf\text{-}rew\ \varphi\ \varphi' \implies is\text{-}dnf\ \varphi'$   
 $\langle proof \rangle$

## 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  $\varphi\ FT$ )  $\varphi$  |*  
*ElimTBFull1'[simp]: elimTBFull (FAnd *FT*  $\varphi$ )  $\varphi$  |*

*ElimTBFull2[simp]: elimTBFull (FAnd  $\varphi\ FF$ ) *FF* |*  
*ElimTBFull2'[simp]: elimTBFull (FAnd *FF*  $\varphi$ ) *FF* |*

$ElimTBFull3[simp]: elimTBFull (FOr \varphi FT) FT \mid$   
 $ElimTBFull3'[simp]: elimTBFull (FOr FT \varphi) FT \mid$   
  
 $ElimTBFull4[simp]: elimTBFull (FOr \varphi FF) \varphi \mid$   
 $ElimTBFull4'[simp]: elimTBFull (FOr FF \varphi) \varphi \mid$   
  
 $ElimTBFull5[simp]: elimTBFull (FNot FT) FF \mid$   
 $ElimTBFull5'[simp]: elimTBFull (FNot FF) FT \mid$   
  
 $ElimTBFull6-l[simp]: elimTBFull (FImp FT \varphi) \varphi \mid$   
 $ElimTBFull6-l'[simp]: elimTBFull (FImp FF \varphi) FT \mid$   
 $ElimTBFull6-r[simp]: elimTBFull (FImp \varphi FT) FT \mid$   
 $ElimTBFull6-r'[simp]: elimTBFull (FImp \varphi FF) (FNot \varphi) \mid$   
  
 $ElimTBFull7-l[simp]: elimTBFull (FEq FT \varphi) \varphi \mid$   
 $ElimTBFull7-l'[simp]: elimTBFull (FEq FF \varphi) (FNot \varphi) \mid$   
 $ElimTBFull7-r[simp]: elimTBFull (FEq \varphi FT) \varphi \mid$   
 $ElimTBFull7-r'[simp]: elimTBFull (FEq \varphi FF) (FNot \varphi) \mid$

The transformation is still consistent.

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

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

**lemma** *no-T-F-symb-except-toplevel-step-exists'*:

**fixes**  $\varphi :: 'v \text{ propo}$

**shows**  $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTBFull } \psi \psi'$

$\langle proof \rangle$

The same applies here. We do not need the assumption, but the deep link between  $\neg \text{no-T-F-except-top-level}$   $\varphi$  and the existence of a rewriting step, still exists.

**lemma** *no-T-F-except-top-level-rew'*:

**fixes**  $\varphi :: 'v \text{ propo}$

**assumes** *noTB*:  $\neg \text{no-T-F-except-top-level } \varphi$

**shows**  $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elimTBFull } \psi \psi'$

$\langle proof \rangle$

**lemma** *elimTBFull-full-propo-rew-step*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$

**assumes** *full* (*propo-rew-step* *elimTBFull*)  $\varphi \psi$

**shows** *no-T-F-except-top-level*  $\psi$

$\langle proof \rangle$

### 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*  $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$   
 $\langle proof \rangle$

```

lemma elim-equiv-inv':
  fixes  $\varphi \psi :: 'v \text{ propo}$ 
  assumes full (propo-rew-step elim-equiv)  $\varphi \psi$  and no-T-F-except-top-level  $\varphi$ 
  shows no-T-F-except-top-level  $\psi$ 
  <proof>

```

```

lemma propo-rew-step-ElimImp-no-T-F: propo-rew-step elim-imp  $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$ 
  <proof>

```

```

lemma elim-imp-inv':
  fixes  $\varphi \psi :: 'v \text{ propo}$ 
  assumes full (propo-rew-step elim-imp)  $\varphi \psi$  and no-T-F-except-top-level  $\varphi$ 
  shows no-T-F-except-top-level  $\psi$ 
  <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  $\Rightarrow$  'a propo  $\Rightarrow$  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'  $\varphi \varphi' \implies \text{is-dnf } \varphi'$ 
  <proof>

```

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

```

definition cnf-rew' :: 'a propo  $\Rightarrow$  'a propo  $\Rightarrow$  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'  $\varphi \varphi' \implies \text{is-cnf } \varphi'$ 
  <proof>

```

```

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

```

## 3.8 Link with Multiset Version

### 3.8.1 Transformation to Multiset

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

**fun** *mset-of-formula* :: 'a propo  $\Rightarrow$  'a literal multiset set **where**  
*mset-of-formula* (*FAnd*  $\varphi$   $\psi$ ) = *mset-of-formula*  $\varphi$   $\cup$  *mset-of-formula*  $\psi$  |  
*mset-of-formula* (*FOr*  $\varphi$   $\psi$ ) = {*mset-of-conj* (*FOr*  $\varphi$   $\psi$ )} |  
*mset-of-formula* (*FVar*  $\psi$ ) = {*mset-of-conj* (*FVar*  $\psi$ )} |  
*mset-of-formula* (*FNot*  $\psi$ ) = {*mset-of-conj* (*FNot*  $\psi$ )} |  
*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*  $\varphi$ )  $\longleftrightarrow$  ( $\exists v. \varphi = \textit{FVar } v \vee \varphi = \textit{FF} \vee \varphi = \textit{FT}$ )  
 $\langle \textit{proof} \rangle$

**lemma** *grouped-by-COr-FNot*:

*grouped-by COr* (*FNot*  $\varphi$ )  $\longleftrightarrow$  ( $\exists v. \varphi = \textit{FVar } v \vee \varphi = \textit{FF} \vee \varphi = \textit{FT}$ )  
 $\langle \textit{proof} \rangle$

**lemma**

**shows** *no-T-F-FF[simp]*:  $\neg \textit{no-T-F FF}$  **and**  
*no-T-F-FT[simp]*:  $\neg \textit{no-T-F FT}$   
 $\langle \textit{proof} \rangle$

**lemma** *grouped-by-CAnd-FAnd*:

*grouped-by CAnd* (*FAnd*  $\varphi_1$   $\varphi_2$ )  $\longleftrightarrow$  *grouped-by CAnd*  $\varphi_1 \wedge$  *grouped-by CAnd*  $\varphi_2$   
 $\langle \textit{proof} \rangle$

**lemma** *grouped-by-COr-FOr*:

*grouped-by COr* (*FOr*  $\varphi_1$   $\varphi_2$ )  $\longleftrightarrow$  *grouped-by COr*  $\varphi_1 \wedge$  *grouped-by COr*  $\varphi_2$   
 $\langle \textit{proof} \rangle$

**lemma** *grouped-by-COr-FAnd[simp]*:  $\neg$  *grouped-by COr* (*FAnd*  $\varphi_1$   $\varphi_2$ )

$\langle \textit{proof} \rangle$

**lemma** *grouped-by-COr-FEq[simp]*:  $\neg$  *grouped-by COr* (*FEq*  $\varphi_1$   $\varphi_2$ )

$\langle \textit{proof} \rangle$

**lemma** [*simp*]:  $\neg$  *grouped-by COr* (*FImp*  $\varphi$   $\psi$ )

$\langle \textit{proof} \rangle$

**lemma** [*simp*]:  $\neg$  *is-conj-with-TF* (*FImp*  $\varphi$   $\psi$ )

$\langle \textit{proof} \rangle$

**lemma** [*simp*]:  $\neg$  *grouped-by COr* (*FEq*  $\varphi$   $\psi$ )

$\langle \textit{proof} \rangle$

**lemma** *[simp]:  $\neg$  is-conj-with-TF (FEq  $\varphi$   $\psi$ )*  
*<proof>*

**lemma** *is-conj-with-TF-Fand:*

*is-conj-with-TF (FAnd  $\varphi1$   $\varphi2$ )  $\implies$  is-conj-with-TF  $\varphi1 \wedge$  is-conj-with-TF  $\varphi2$*   
*<proof>*

**lemma** *is-conj-with-TF-FOr:*

*is-conj-with-TF (FOr  $\varphi1$   $\varphi2$ )  $\implies$  grouped-by COr  $\varphi1 \wedge$  grouped-by COr  $\varphi2$*   
*<proof>*

**lemma** *grouped-by-COr-mset-of-formula:*

*grouped-by COr  $\varphi \implies$  mset-of-formula  $\varphi =$  (if  $\varphi = FT$  then  $\{\}$  else  $\{mset-of-conj \varphi\}$ )*  
*<proof>*

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

**theorem**

**fixes**  $\varphi :: 'v$  *propo*

**assumes** *is-cnf  $\varphi$*

**shows** *eval A  $\varphi \longleftrightarrow$  Partial-Clausal-Logic.true-cls ( $\{Pos\ v|v. A\ v\} \cup \{Neg\ v|v. \neg A\ v\}$ )*  
*(mset-of-formula  $\varphi$ )*

*<proof>*

**end**

**theory** *Prop-Resolution*

**imports** *Partial-Clausal-Logic List-More Wellfounded-More*

**begin**





# Chapter 4

## Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

### 4.1 Resolution

#### 4.1.1 Simplification Rules

**inductive** *simplify* :: 'v clauses  $\Rightarrow$  'v clauses  $\Rightarrow$  bool **for** *N* :: 'v clause set **where**

*tautology-deletion*:

$A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \} \in N \implies simplify\ N\ (N - \{A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\})$

*condensation*:

$A + \{\#L\# \} + \{\#L\# \} \in N \implies simplify\ N\ (N - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\})$

*subsumption*:

$A \in N \implies A \subset\# B \implies B \in N \implies simplify\ N\ (N - \{B\})$

**lemma** *simplify-preserves-un-sat'*:

**fixes** *N N'* :: 'v clauses

**assumes** *simplify N N'*

**and** *total-over-m I N*

**shows**  $I \models_s N' \longrightarrow I \models_s N$

*<proof>*

**lemma** *simplify-preserves-un-sat*:

**fixes** *N N'* :: 'v clauses

**assumes** *simplify N N'*

**and** *total-over-m I N*

**shows**  $I \models_s N \longrightarrow I \models_s N'$

*<proof>*

**lemma** *simplify-preserves-un-sat''*:

**fixes** *N N'* :: 'v clauses

**assumes** *simplify N N'*

**and** *total-over-m I N'*

**shows**  $I \models_s N \longrightarrow I \models_s N'$

*<proof>*

**lemma** *simplify-preserves-un-sat-eq*:

**fixes** *N N'* :: 'v clauses

**assumes** *simplify N N'*

**and** *total-over-m I N*

**shows**  $I \models_s N \longleftrightarrow I \models_s N'$

$\langle \text{proof} \rangle$

**lemma** *simplify-preserves-finite*:

**assumes** *simplify*  $\psi \ \psi'$

**shows** *finite*  $\psi \longleftrightarrow \text{finite } \psi'$

$\langle \text{proof} \rangle$

**lemma** *rtranclp-simplify-preserves-finite*:

**assumes** *rtranclp simplify*  $\psi \ \psi'$

**shows** *finite*  $\psi \longleftrightarrow \text{finite } \psi'$

$\langle \text{proof} \rangle$

**lemma** *simplify-atms-of-ms*:

**assumes** *simplify*  $\psi \ \psi'$

**shows** *atms-of-ms*  $\psi' \subseteq \text{atms-of-ms } \psi$

$\langle \text{proof} \rangle$

**lemma** *rtranclp-simplify-atms-of-ms*:

**assumes** *rtranclp simplify*  $\psi \ \psi'$

**shows** *atms-of-ms*  $\psi' \subseteq \text{atms-of-ms } \psi$

$\langle \text{proof} \rangle$

**lemma** *factoring-imp-simplify*:

**assumes**  $\{\#L\# \} + \{\#L\# \} + C \in N$

**shows**  $\exists N'. \text{ simplify } N \ N'$

$\langle \text{proof} \rangle$

## 4.1.2 Unconstrained Resolution

**type-synonym** *'v uncon-state* = *'v clauses*

**inductive** *uncon-res* :: *'v uncon-state*  $\Rightarrow$  *'v uncon-state*  $\Rightarrow$  *bool* **where**

*resolution*:

$\{\#Pos \ p\# \} + C \in N \implies \{\#Neg \ p\# \} + D \in N \implies (\{\#Pos \ p\# \} + C, \{\#Neg \ p\# \} + D) \notin$   
*already-used*

$\implies \text{uncon-res } (N) (N \cup \{C + D\}) \mid$

*factoring*:  $\{\#L\# \} + \{\#L\# \} + C \in N \implies \text{uncon-res } N (N \cup \{C + \{\#L\# \}\})$

**lemma** *uncon-res-increasing*:

**assumes** *uncon-res*  $S \ S'$  **and**  $\psi \in S$

**shows**  $\psi \in S'$

$\langle \text{proof} \rangle$

**lemma** *rtranclp-uncon-inference-increasing*:

**assumes** *rtranclp uncon-res*  $S \ S'$  **and**  $\psi \in S$

**shows**  $\psi \in S'$

$\langle \text{proof} \rangle$

## Subsumption

**definition** *subsumes* :: *'a literal multiset*  $\Rightarrow$  *'a literal multiset*  $\Rightarrow$  *bool* **where**

*subsumes*  $\chi \ \chi' \longleftrightarrow$

$(\forall I. \text{total-over-m } I \ \{\chi'\} \longrightarrow \text{total-over-m } I \ \{\chi\})$

$\wedge (\forall I. \text{total-over-m } I \ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi')$

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

*subsumes*  $\chi \ \chi$

$\langle \text{proof} \rangle$

**lemma** *subsumes-subsumption*:

**assumes** *subsumes*  $D \chi$   
**and**  $C \subset\# D$  **and**  $\neg \text{tautology } \chi$   
**shows** *subsumes*  $C \chi$   $\langle \text{proof} \rangle$

**lemma** *subsumes-tautology*:

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

### 4.1.3 Inference Rule

**type-synonym** *'v state* = *'v clauses*  $\times$  (*'v clause*  $\times$  *'v clause*) *set*

**inductive** *inference-clause* :: *'v state*  $\Rightarrow$  *'v clause*  $\times$  (*'v clause*  $\times$  *'v clause*) *set*  $\Rightarrow$  *bool*

(**infix**  $\Rightarrow_{\text{Res}}$  100) **where**

*resolution*:

$\{\# \text{Pos } p\# \} + C \in N \implies \{\# \text{Neg } p\# \} + D \in N \implies (\{\# \text{Pos } p\# \} + C, \{\# \text{Neg } p\# \} + D) \notin$   
*already-used*

$\implies \text{inference-clause } (N, \text{already-used}) (C + D, \text{already-used} \cup \{(\{\# \text{Pos } p\# \} + C, \{\# \text{Neg } p\# \} + D)\}) \mid$

*factoring*:  $\{\# L\# \} + \{\# L\# \} + C \in N \implies \text{inference-clause } (N, \text{already-used}) (C + \{\# L\# \}, \text{already-used})$

**inductive** *inference* :: *'v state*  $\Rightarrow$  *'v state*  $\Rightarrow$  *bool* **where**

*inference-step*: *inference-clause*  $S$  (*clause*, *already-used*)

$\implies \text{inference } S (\text{fst } S \cup \{\text{clause}\}, \text{already-used})$

**abbreviation** *already-used-inv*

:: *'a literal multiset set*  $\times$  (*'a literal multiset*  $\times$  *'a literal multiset*) *set*  $\Rightarrow$  *bool* **where**

*already-used-inv state*  $\equiv$

$(\forall (A, B) \in \text{snd state}. \exists p. \text{Pos } p \in\# A \wedge \text{Neg } p \in\# B \wedge$   
 $((\exists \chi \in \text{fst state}. \text{subsumes } \chi ((A - \{\# \text{Pos } p\# \}) + (B - \{\# \text{Neg } p\# \})))$   
 $\vee \text{tautology } ((A - \{\# \text{Pos } p\# \}) + (B - \{\# \text{Neg } p\# \}))))$

**lemma** *inference-clause-preserves-already-used-inv*:

**assumes** *inference-clause*  $S S'$

**and** *already-used-inv*  $S$

**shows** *already-used-inv*  $(\text{fst } S \cup \{\text{fst } S'\}, \text{snd } S')$

$\langle \text{proof} \rangle$

**lemma** *inference-preserves-already-used-inv*:

**assumes** *inference*  $S S'$

**and** *already-used-inv*  $S$

**shows** *already-used-inv*  $S'$

$\langle \text{proof} \rangle$

**lemma** *rtranclp-inference-preserves-already-used-inv*:

**assumes** *rtranclp inference*  $S S'$

**and** *already-used-inv*  $S$

**shows** *already-used-inv*  $S'$

$\langle \text{proof} \rangle$

**lemma** *subsumes-condensation*:

**assumes** *subsumes*  $(C + \{\#L\# \} + \{\#L\# \}) D$   
**shows** *subsumes*  $(C + \{\#L\# \}) D$   
 $\langle \text{proof} \rangle$

**lemma** *simplify-preserves-already-used-inv*:  
**assumes** *simplify*  $N N'$   
**and** *already-used-inv*  $(N, \text{already-used})$   
**shows** *already-used-inv*  $(N', \text{already-used})$   
 $\langle \text{proof} \rangle$

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

**lemma** *inference-increasing*:  
**assumes** *inference*  $S S'$  **and**  $\psi \in \text{fst } S$   
**shows**  $\psi \in \text{fst } S'$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-inference-increasing*:  
**assumes** *rtranclp inference*  $S S'$  **and**  $\psi \in \text{fst } S$   
**shows**  $\psi \in \text{fst } S'$   
 $\langle \text{proof} \rangle$

**lemma** *inference-clause-already-used-increasing*:  
**assumes** *inference-clause*  $S S'$   
**shows**  $\text{snd } S \subseteq \text{snd } S'$   
 $\langle \text{proof} \rangle$

**lemma** *inference-already-used-increasing*:  
**assumes** *inference*  $S S'$   
**shows**  $\text{snd } S \subseteq \text{snd } S'$   
 $\langle \text{proof} \rangle$

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

**lemma** *inference-preserves-un-sat*:  
**fixes**  $N N' :: 'v \text{ clauses}$   
**assumes** *inference*  $T T'$   
**and** *total-over-m*  $I (\text{fst } T)$   
**and** *consistent*: *consistent-interp*  $I$   
**shows**  $I \models_s \text{fst } T \longleftrightarrow I \models_s \text{fst } T'$   
 $\langle \text{proof} \rangle$

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

**lemma** *inference-preserves-atms-of-ms*:  
**fixes**  $N\ N' :: 'v \text{ clauses}$   
**assumes** *inference*  $T\ T'$   
**shows**  $\text{atms-of-ms } (\text{fst } T') \subseteq \text{atms-of-ms } (\text{fst } T)$   
 $\langle \text{proof} \rangle$

**lemma** *inference-preserves-total*:  
**fixes**  $N\ N' :: 'v \text{ clauses}$   
**assumes** *inference*  $(N, \text{already-used})\ (N', \text{already-used}')$   
**shows**  $\text{total-over-m } I\ N \implies \text{total-over-m } I\ N'$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-inference-preserves-total*:  
**assumes** *rtranclp inference*  $T\ T'$   
**shows**  $\text{total-over-m } I\ (\text{fst } T) \implies \text{total-over-m } I\ (\text{fst } T')$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-inference-preserves-un-sat*:  
**assumes** *rtranclp inference*  $N\ N'$   
**and**  $\text{total-over-m } I\ (\text{fst } N)$   
**and** *consistent: consistent-interp*  $I$   
**shows**  $I \models_s \text{fst } N \longleftrightarrow I \models_s \text{fst } N'$   
 $\langle \text{proof} \rangle$

**lemma** *inference-preserves-finite*:  
**assumes** *inference*  $\psi\ \psi'$  **and** *finite*  $(\text{fst } \psi)$   
**shows** *finite*  $(\text{fst } \psi')$   
 $\langle \text{proof} \rangle$

**lemma** *inference-clause-preserves-finite-snd*:  
**assumes** *inference-clause*  $\psi\ \psi'$  **and** *finite*  $(\text{snd } \psi)$   
**shows** *finite*  $(\text{snd } \psi')$   
 $\langle \text{proof} \rangle$

**lemma** *inference-preserves-finite-snd*:  
**assumes** *inference*  $\psi\ \psi'$  **and** *finite*  $(\text{snd } \psi)$   
**shows** *finite*  $(\text{snd } \psi')$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-inference-preserves-finite*:  
**assumes** *rtranclp inference*  $\psi\ \psi'$  **and** *finite*  $(\text{fst } \psi)$   
**shows** *finite*  $(\text{fst } \psi')$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

**lemma** *simplify-preserve-partial-leaf*:  
*simplify*  $N \ N' \implies \text{partial-interps } \text{Leaf } I \ N \implies \text{partial-interps } \text{Leaf } I \ N'$   
 ⟨*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*  $t \ I \ (\text{fst } S)$   
**shows** *partial-interps*  $t \ I \ (\text{fst } S')$   
 ⟨*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')*  
 $\langle \text{proof} \rangle$

**function** *build-sem-tree* :: '*v* :: linorder set  $\Rightarrow$  '*v* clauses  $\Rightarrow$  '*v* sem-tree **where**

*build-sem-tree atms  $\psi$  =*  
*(if atms = {}  $\vee$   $\neg$  finite atms*  
*then Leaf*  
*else Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )*  
*(build-sem-tree (Set.remove (Min atms) atms)  $\psi$ ))*

$\langle \text{proof} \rangle$

**termination**

$\langle \text{proof} \rangle$

**declare** *build-sem-tree.induct[case-names tree]*

**lemma** *unsatisfiable-empty[simp]*:

$\neg \text{unsatisfiable } \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *partial-interps-build-sem-tree-atms-general*:

**fixes**  $\psi :: 'v :: \text{linorder clauses}$  **and**  $p :: 'v \text{ literal list}$   
**assumes** *unsat: unsatisfiable  $\psi$*  **and** *finite  $\psi$*  **and** *consistent-interp I*  
**and** *finite atms*  
**and** *atms-of-ms  $\psi = \text{atms} \cup \text{atms-of-s } I$*  **and**  *$\text{atms} \cap \text{atms-of-s } I = \{\}$*   
**shows** *partial-interps (build-sem-tree atms  $\psi$ ) I  $\psi$*   
 $\langle \text{proof} \rangle$

**lemma** *partial-interps-build-sem-tree-atms*:

**fixes**  $\psi :: 'v :: \text{linorder clauses}$  **and**  $p :: 'v \text{ literal list}$   
**assumes** *unsat: unsatisfiable  $\psi$*  **and** *finite: finite  $\psi$*   
**shows** *partial-interps (build-sem-tree (atms-of-ms  $\psi$ )  $\psi$ ) {}  $\psi$*   
 $\langle \text{proof} \rangle$

**lemma** *can-decrease-count*:

**fixes**  $\psi'' :: 'v \text{ clauses} \times ('v \text{ clause} \times 'v \text{ clause} \times 'v) \text{ set}$   
**assumes** *count  $\chi L = n$*   
**and**  *$L \in \# \chi$*  **and**  *$\chi \in \text{fst } \psi$*   
**shows**  $\exists \psi' \chi'. \text{inference}^{**} \psi \psi' \wedge \chi' \in \text{fst } \psi' \wedge (\forall L. L \in \# \chi \longleftrightarrow L \in \# \chi')$   
 $\wedge \text{count } \chi' L = 1$   
 $\wedge (\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi')$   
 $\wedge (I \models \chi \longleftrightarrow I \models \chi')$   
 $\wedge (\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi'\})$

$\langle \text{proof} \rangle$

**lemma** *can-decrease-tree-size*:

**fixes**  $\psi :: 'v \text{ state}$  **and**  $\text{tree} :: 'v \text{ sem-tree}$   
**assumes** *finite (fst  $\psi$ )* **and** *already-used-inv  $\psi$*   
**and** *partial-interps tree I (fst  $\psi$ )*  
**shows**  $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{inference}^{**} \psi \psi' \wedge \text{partial-interps tree}' I (\text{fst } \psi')$   
 $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$

$\langle \text{proof} \rangle$

**lemma** *inference-completeness-inv*:  
**fixes**  $\psi :: 'v :: \text{linorder state}$   
**assumes**  
    $\text{unsat}: \neg \text{satisfiable } (\text{fst } \psi)$  **and**  
    $\text{finite}: \text{finite } (\text{fst } \psi)$  **and**  
    $\text{a-u-v}: \text{already-used-inv } \psi$   
**shows**  $\exists \psi'. (\text{inference}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$   
 $\langle \text{proof} \rangle$

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

**lemma** *inference-soundness*:  
**fixes**  $\psi :: 'v :: \text{linorder state}$   
**assumes**  $\text{rtranclp inference } \psi \psi' \text{ and } \{\#\} \in \text{fst } \psi'$   
**shows**  $\text{unsatisfiable } (\text{fst } \psi)$   
 $\langle \text{proof} \rangle$

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

#### 4.1.4 Lemma about the simplified state

**abbreviation**  $\text{simplified } \psi \equiv (\text{no-step simplify } \psi)$

**lemma** *simplified-count*:  
**assumes**  $\text{simp}: \text{simplified } \psi \text{ and } \chi: \chi \in \psi$   
**shows**  $\text{count } \chi L \leq 1$   
 $\langle \text{proof} \rangle$

**lemma** *simplified-no-both*:  
**assumes**  $\text{simp}: \text{simplified } \psi \text{ and } \chi: \chi \in \psi$   
**shows**  $\neg (L \in \# \chi \wedge \neg L \in \# \chi)$   
 $\langle \text{proof} \rangle$

**lemma** *simplified-not-tautology*:  
**assumes**  $\text{simplified } \{\psi\}$   
**shows**  $\sim \text{tautology } \psi$   
 $\langle \text{proof} \rangle$

**lemma** *simplified-remove*:  
**assumes**  $\text{simplified } \{\psi\}$   
**shows**  $\text{simplified } \{\psi - \{\#l\# \}\}$   
 $\langle \text{proof} \rangle$

**lemma** *in-simplified-simplified*:



**assumes** *simp: simplified  $\psi$*  **and** *incl:  $\psi' \subseteq \psi$*   
**shows** *simplified  $\psi'$*   
 $\langle proof \rangle$

**lemma** *simplified-in:*  
**assumes** *simplified  $\psi$*   
**and**  *$N \in \psi$*   
**shows** *simplified  $\{N\}$*   
 $\langle proof \rangle$

**lemma** *subsumes-imp-formula:*  
**assumes**  *$\psi \leq \# \varphi$*   
**shows**  *$\{\psi\} \models_p \varphi$*   
 $\langle proof \rangle$

**lemma** *simplified-imp-distinct-mset-tauto:*  
**assumes** *simp: simplified  $\psi'$*   
**shows** *distinct-mset-set  $\psi'$  and  $\forall \chi \in \psi'. \neg \text{tautology } \chi$*   
 $\langle proof \rangle$

**lemma** *simplified-no-more-full1-simplified:*  
**assumes** *simplified  $\psi$*   
**shows**  *$\neg \text{full1 simplify } \psi \psi'$*   
 $\langle proof \rangle$

#### 4.1.5 Resolution and Invariants

**inductive** *resolution* :: *'v state  $\Rightarrow$  'v state  $\Rightarrow$  bool* **where**  
*full1-simp: full1 simplify  $N N' \Rightarrow$  resolution  $(N, \text{already-used}) (N', \text{already-used})$  |*  
*inferring: inference  $(N, \text{already-used}) (N', \text{already-used}') \Rightarrow$  simplified  $N$*   
 *$\Rightarrow$  full simplify  $N' N'' \Rightarrow$  resolution  $(N, \text{already-used}) (N'', \text{already-used}')$*

##### Invariants

**lemma** *resolution-finite:*  
**assumes** *resolution  $\psi \psi'$  and finite  $(fst \psi)$*   
**shows** *finite  $(fst \psi')$*   
 $\langle proof \rangle$

**lemma** *rtrancp-resolution-finite:*  
**assumes** *resolution\*\*  $\psi \psi'$  and finite  $(fst \psi)$*   
**shows** *finite  $(fst \psi')$*   
 $\langle proof \rangle$

**lemma** *resolution-finite-snd:*  
**assumes** *resolution  $\psi \psi'$  and finite  $(snd \psi)$*   
**shows** *finite  $(snd \psi')$*   
 $\langle proof \rangle$

**lemma** *rtrancp-resolution-finite-snd:*  
**assumes** *resolution\*\*  $\psi \psi'$  and finite  $(snd \psi)$*   
**shows** *finite  $(snd \psi')$*   
 $\langle proof \rangle$

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

**shows** *simplified* (fst  $\psi'$ )  
 <proof>

**lemma** *trancpl-resolution-always-simplified*:

**assumes** *trancpl resolution*  $\psi \psi'$   
**shows** *simplified* (fst  $\psi'$ )  
 <proof>

**lemma** *resolution-atms-of*:

**assumes** *resolution*  $\psi \psi'$  **and** *finite* (fst  $\psi$ )  
**shows** *atms-of-ms* (fst  $\psi'$ )  $\subseteq$  *atms-of-ms* (fst  $\psi$ )  
 <proof>

**lemma** *rtrancpl-resolution-atms-of*:

**assumes** *resolution*\*\*  $\psi \psi'$  **and** *finite* (fst  $\psi$ )  
**shows** *atms-of-ms* (fst  $\psi'$ )  $\subseteq$  *atms-of-ms* (fst  $\psi$ )  
 <proof>

**lemma** *resolution-include*:

**assumes** *res: resolution*  $\psi \psi'$  **and** *finite: finite* (fst  $\psi$ )  
**shows** *fst*  $\psi' \subseteq$  *simple-clss* (*atms-of-ms* (fst  $\psi$ ))  
 <proof>

**lemma** *rtrancpl-resolution-include*:

**assumes** *res: trancpl resolution*  $\psi \psi'$  **and** *finite: finite* (fst  $\psi$ )  
**shows** *fst*  $\psi' \subseteq$  *simple-clss* (*atms-of-ms* (fst  $\psi$ ))  
 <proof>

**abbreviation** *already-used-all-simple*

$:: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$  **where**  
*already-used-all-simple* *already-used* *vars*  $\equiv$   
 $(\forall (A, B) \in \text{already-used. simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars})$

**lemma** *already-used-all-simple-vars-incl*:

**assumes** *vars*  $\subseteq$  *vars'*  
**shows** *already-used-all-simple* *a vars*  $\implies$  *already-used-all-simple* *a vars'*  
 <proof>

**lemma** *inference-clause-preserves-already-used-all-simple*:

**assumes** *inference-clause*  $S S'$   
**and** *already-used-all-simple* (snd  $S$ ) *vars*  
**and** *simplified* (fst  $S$ )  
**and** *atms-of-ms* (fst  $S$ )  $\subseteq$  *vars*  
**shows** *already-used-all-simple* (snd (fst  $S \cup \{\text{fst } S'\}, \text{snd } S'$ )) *vars*  
 <proof>

**lemma** *inference-preserves-already-used-all-simple*:

**assumes** *inference*  $S S'$   
**and** *already-used-all-simple* (snd  $S$ ) *vars*  
**and** *simplified* (fst  $S$ )  
**and** *atms-of-ms* (fst  $S$ )  $\subseteq$  *vars*  
**shows** *already-used-all-simple* (snd  $S'$ ) *vars*  
 <proof>

**lemma** *already-used-all-simple-inv*:

**assumes** *resolution*  $S S'$

**and** *already-used-all-simple* (*snd S*) *vars*  
**and** *atms-of-ms* (*fst S*)  $\subseteq$  *vars*  
**shows** *already-used-all-simple* (*snd S'*) *vars*  
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-already-used-all-simple-inv*:  
**assumes** *resolution\*\* S S'*  
**and** *already-used-all-simple* (*snd S*) *vars*  
**and** *atms-of-ms* (*fst S*)  $\subseteq$  *vars*  
**and** *finite* (*fst S*)  
**shows** *already-used-all-simple* (*snd S'*) *vars*  
 $\langle \text{proof} \rangle$

**lemma** *inference-clause-simplified-already-used-subset*:  
**assumes** *inference-clause S S'*  
**and** *simplified* (*fst S*)  
**shows** *snd S*  $\subset$  *snd S'*  
 $\langle \text{proof} \rangle$

**lemma** *inference-simplified-already-used-subset*:  
**assumes** *inference S S'*  
**and** *simplified* (*fst S*)  
**shows** *snd S*  $\subset$  *snd S'*  
 $\langle \text{proof} \rangle$

**lemma** *resolution-simplified-already-used-subset*:  
**assumes** *resolution S S'*  
**and** *simplified* (*fst S*)  
**shows** *snd S*  $\subset$  *snd S'*  
 $\langle \text{proof} \rangle$

**lemma** *trancp-resolution-simplified-already-used-subset*:  
**assumes** *trancp resolution S S'*  
**and** *simplified* (*fst S*)  
**shows** *snd S*  $\subset$  *snd S'*  
 $\langle \text{proof} \rangle$

**abbreviation** *already-used-top vars*  $\equiv$  *simple-clss vars*  $\times$  *simple-clss vars*

**lemma** *already-used-all-simple-in-already-used-top*:  
**assumes** *already-used-all-simple s vars* **and** *finite vars*  
**shows** *s*  $\subseteq$  *already-used-top vars*  
 $\langle \text{proof} \rangle$

**lemma** *already-used-top-finite*:  
**assumes** *finite vars*  
**shows** *finite* (*already-used-top vars*)  
 $\langle \text{proof} \rangle$

**lemma** *already-used-top-increasing*:  
**assumes** *var*  $\subseteq$  *var'* **and** *finite var'*  
**shows** *already-used-top var*  $\subseteq$  *already-used-top var'*  
 $\langle \text{proof} \rangle$

**lemma** *already-used-all-simple-finite*:  
**fixes** *s* :: ('a literal multiset  $\times$  'a literal multiset) *set* **and** *vars* :: 'a *set*

**assumes** *already-used-all-simple s vars* **and** *finite vars*  
**shows** *finite s*  
 $\langle \text{proof} \rangle$

**abbreviation** *card-simple vars*  $\psi \equiv \text{card } (\text{already-used-top vars} - \psi)$

**lemma** *resolution-card-simple-decreasing:*

**assumes** *res: resolution  $\psi \psi'$*   
**and** *a-u-s: already-used-all-simple (snd  $\psi$ ) vars*  
**and** *finite-v: finite vars*  
**and** *finite-fst: finite (fst  $\psi$ )*  
**and** *finite-snd: finite (snd  $\psi$ )*  
**and** *simp: simplified (fst  $\psi$ )*  
**and** *atms-of-ms (fst  $\psi$ )  $\subseteq$  vars*  
**shows** *card-simple vars (snd  $\psi'$ )  $<$  card-simple vars (snd  $\psi$ )*  
 $\langle \text{proof} \rangle$

**lemma** *trancp-resolution-card-simple-decreasing:*

**assumes** *trancp resolution  $\psi \psi'$  and finite-fst: finite (fst  $\psi$ )*  
**and** *already-used-all-simple (snd  $\psi$ ) vars*  
**and** *atms-of-ms (fst  $\psi$ )  $\subseteq$  vars*  
**and** *finite-v: finite vars*  
**and** *finite-snd: finite (snd  $\psi$ )*  
**and** *simplified (fst  $\psi$ )*  
**shows** *card-simple vars (snd  $\psi'$ )  $<$  card-simple vars (snd  $\psi$ )*  
 $\langle \text{proof} \rangle$

**lemma** *trancp-resolution-card-simple-decreasing-2:*

**assumes** *trancp resolution  $\psi \psi'$*   
**and** *finite-fst: finite (fst  $\psi$ )*  
**and** *empty-snd: snd  $\psi = \{\}$*   
**and** *simplified (fst  $\psi$ )*  
**shows** *card-simple (atms-of-ms (fst  $\psi$ )) (snd  $\psi'$ )  $<$  card-simple (atms-of-ms (fst  $\psi$ )) (snd  $\psi$ )*  
 $\langle \text{proof} \rangle$

## well-foundness if the relation

**lemma** *wf-simplified-resolution:*

**assumes** *f-vars: finite vars*  
**shows** *wf  $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms (fst } x) \subseteq \text{vars} \wedge \text{simplified (fst } x) \wedge \text{finite (snd } x) \wedge \text{finite (fst } x) \wedge \text{already-used-all-simple (snd } x) \text{ vars}) \wedge \text{resolution } x y\}$*   
 $\langle \text{proof} \rangle$

**lemma** *wf-simplified-resolution':*

**assumes** *f-vars: finite vars*  
**shows** *wf  $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms (fst } x) \subseteq \text{vars} \wedge \neg \text{simplified (fst } x) \wedge \text{finite (snd } x) \wedge \text{finite (fst } x) \wedge \text{already-used-all-simple (snd } x) \text{ vars}) \wedge \text{resolution } x y\}$*   
 $\langle \text{proof} \rangle$

**lemma** *wf-resolution:*

**assumes** *f-vars: finite vars*  
**shows** *wf  $(\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms (fst } x) \subseteq \text{vars} \wedge \text{simplified (fst } x) \wedge \text{finite (snd } x) \wedge \text{finite (fst } x) \wedge \text{already-used-all-simple (snd } x) \text{ vars}) \wedge \text{resolution } x y\} \cup \{(y, x). (\text{atms-of-ms (fst } x) \subseteq \text{vars} \wedge \neg \text{simplified (fst } x) \wedge \text{finite (snd } x) \wedge \text{finite (fst } x) \wedge \text{already-used-all-simple (snd } x) \text{ vars}) \wedge \text{resolution } x y\})$*

$\wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x \ y\} \text{ (is wf } (?R \cup ?S))$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-simplify-already-used-inv*:  
**assumes** *simplify\*\**  $S \ S'$   
**and** *already-used-inv*  $(S, N)$   
**shows** *already-used-inv*  $(S', N)$   
 $\langle \text{proof} \rangle$

**lemma** *full1-simplify-already-used-inv*:  
**assumes** *full1 simplify*  $S \ S'$   
**and** *already-used-inv*  $(S, N)$   
**shows** *already-used-inv*  $(S', N)$   
 $\langle \text{proof} \rangle$

**lemma** *full-simplify-already-used-inv*:  
**assumes** *full simplify*  $S \ S'$   
**and** *already-used-inv*  $(S, N)$   
**shows** *already-used-inv*  $(S', N)$   
 $\langle \text{proof} \rangle$

**lemma** *resolution-already-used-inv*:  
**assumes** *resolution*  $S \ S'$   
**and** *already-used-inv*  $S$   
**shows** *already-used-inv*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-resolution-already-used-inv*:  
**assumes** *resolution\*\**  $S \ S'$   
**and** *already-used-inv*  $S$   
**shows** *already-used-inv*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *rtanclp-simplify-preserves-unsat*:  
**assumes** *simplify\*\**  $\psi \ \psi'$   
**shows** *satisfiable*  $\psi' \longrightarrow \text{satisfiable } \psi$   
 $\langle \text{proof} \rangle$

**lemma** *full1-simplify-preserves-unsat*:  
**assumes** *full1 simplify*  $\psi \ \psi'$   
**shows** *satisfiable*  $\psi' \longrightarrow \text{satisfiable } \psi$   
 $\langle \text{proof} \rangle$

**lemma** *full-simplify-preserves-unsat*:  
**assumes** *full simplify*  $\psi \ \psi'$   
**shows** *satisfiable*  $\psi' \longrightarrow \text{satisfiable } \psi$   
 $\langle \text{proof} \rangle$

**lemma** *resolution-preserves-unsat*:  
**assumes** *resolution*  $\psi \ \psi'$   
**shows** *satisfiable*  $(\text{fst } \psi') \longrightarrow \text{satisfiable } (\text{fst } \psi)$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-resolution-preserves-unsat*:  
**assumes** *resolution\*\**  $\psi \ \psi'$   
**shows** *satisfiable*  $(\text{fst } \psi') \longrightarrow \text{satisfiable } (\text{fst } \psi)$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-simplify-preserve-partial-tree*:  
**assumes** *simplify\*\* N N'*  
**and** *partial-interps t I N*  
**shows** *partial-interps t I N'*  
 $\langle \text{proof} \rangle$

**lemma** *full1-simplify-preserve-partial-tree*:  
**assumes** *full1 simplify N N'*  
**and** *partial-interps t I N*  
**shows** *partial-interps t I N'*  
 $\langle \text{proof} \rangle$

**lemma** *full-simplify-preserve-partial-tree*:  
**assumes** *full simplify N N'*  
**and** *partial-interps t I N*  
**shows** *partial-interps t I N'*  
 $\langle \text{proof} \rangle$

**lemma** *resolution-preserve-partial-tree*:  
**assumes** *resolution S S'*  
**and** *partial-interps t I (fst S)*  
**shows** *partial-interps t I (fst S')*  
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-resolution-preserve-partial-tree*:  
**assumes** *resolution\*\* S S'*  
**and** *partial-interps t I (fst S)*  
**shows** *partial-interps t I (fst S')*  
 $\langle \text{proof} \rangle$   
**thm** *nat-less-induct nat.induct*

**lemma** *nat-ge-induct[case-names 0 Suc]*:  
**assumes** *P 0*  
**and**  $\bigwedge n. (\bigwedge m. m < \text{Suc } n \implies P m) \implies P (\text{Suc } n)$   
**shows** *P n*  
 $\langle \text{proof} \rangle$

**lemma** *wf-always-more-step-False*:  
**assumes** *wf R*  
**shows**  $(\forall x. \exists z. (z, x) \in R) \implies \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *finite-finite-mset-element-of-mset[simp]*:  
**assumes** *finite N*  
**shows** *finite  $\{f \varphi L \mid \varphi L. \varphi \in N \wedge L \in \# \varphi \wedge P \varphi L\}$*   
 $\langle \text{proof} \rangle$

**definition** *sum-count-ge-2* :: *'a multiset set  $\Rightarrow$  nat  $(\Xi)$  where*  
*sum-count-ge-2  $\equiv$  folding.F  $(\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\}))$  0*

**interpretation** *sum-count-ge-2*:  
*folding  $(\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\}))$  0*  
**rewrites**

*folding.F* ( $\lambda\varphi. op + (msetsum \{\#count \varphi L \mid L \in \# \varphi. 2 \leq count \varphi L\#\}) 0 = sum-count-ge-2$ )  
 <proof>

**lemma** *finite-incl-le-setsum*:

*finite* ( $B :: 'a \text{ multiset set}$ )  $\implies A \subseteq B \implies \Xi A \leq \Xi B$   
 <proof>

**lemma** *simplify-finite-measure-decrease*:

*simplify*  $N N' \implies \text{finite } N \implies card N' + \Xi N' < card N + \Xi N$   
 <proof>

**lemma** *simplify-terminates*:

*wf*  $\{(N', N). \text{finite } N \wedge \text{simplify } N N'\}$   
 <proof>

**lemma** *wf-terminates*:

**assumes** *wf*  $r$   
**shows**  $\exists N'. (N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r)$   
 <proof>

**lemma** *rtranclp-simplify-terminates*:

**assumes** *fin*: *finite*  $N$   
**shows**  $\exists N'. \text{simplify}^{**} N N' \wedge \text{simplified } N'$   
 <proof>

**lemma** *finite-simplified-full1-simp*:

**assumes** *finite*  $N$   
**shows**  $\text{simplified } N \vee (\exists N'. \text{full1 simplify } N N')$   
 <proof>

**lemma** *finite-simplified-full-simp*:

**assumes** *finite*  $N$   
**shows**  $\exists N'. \text{full simplify } N N'$   
 <proof>

**lemma** *can-decrease-tree-size-resolution*:

**fixes**  $\psi :: 'v \text{ state}$  **and**  $\text{tree} :: 'v \text{ sem-tree}$   
**assumes** *finite* (*fst*  $\psi$ ) **and** *already-used-inv*  $\psi$   
**and** *partial-interps*  $\text{tree } I$  (*fst*  $\psi$ )  
**and** *simplified* (*fst*  $\psi$ )  
**shows**  $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{resolution}^{**} \psi \psi' \wedge \text{partial-interps } \text{tree}' I$  (*fst*  $\psi'$ )  
 $\wedge (\text{sem-tree-size } \text{tree}' < \text{sem-tree-size } \text{tree} \vee \text{sem-tree-size } \text{tree} = 0)$   
 <proof>

**lemma** *resolution-completeness-inv*:

**fixes**  $\psi :: 'v :: \text{linorder state}$   
**assumes**  
*unsat*:  $\neg \text{satisfiable}$  (*fst*  $\psi$ ) **and**  
*finite*: *finite* (*fst*  $\psi$ ) **and**  
*a-u-v*: *already-used-inv*  $\psi$   
**shows**  $\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$   
 <proof>

**lemma** *resolution-preserves-already-used-inv*:

**assumes** *resolution*  $S\ S'$   
**and** *already-used-inv*  $S$   
**shows** *already-used-inv*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-resolution-preserves-already-used-inv*:  
**assumes** *resolution\*\**  $S\ S'$   
**and** *already-used-inv*  $S$   
**shows** *already-used-inv*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *resolution-completeness*:  
**fixes**  $\psi :: 'v :: \text{linorder state}$   
**assumes** *unsat*:  $\neg \text{satisfiable } (\text{fst } \psi)$   
**and** *finite*: *finite*  $(\text{fst } \psi)$   
**and** *snd*  $\psi = \{\}$   
**shows**  $\exists \psi'. (\text{resolution** } \psi\ \psi' \wedge \{\#\} \in \text{fst } \psi')$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-preserves-sat*:  
**assumes** *simplify\*\**  $S\ S'$   
**and** *satisfiable*  $S$   
**shows** *satisfiable*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *resolution-preserves-sat*:  
**assumes** *resolution*  $S\ S'$   
**and** *satisfiable*  $(\text{fst } S)$   
**shows** *satisfiable*  $(\text{fst } S')$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-resolution-preserves-sat*:  
**assumes** *resolution\*\**  $S\ S'$   
**and** *satisfiable*  $(\text{fst } S)$   
**shows** *satisfiable*  $(\text{fst } S')$   
 $\langle \text{proof} \rangle$

**lemma** *resolution-soundness*:  
**fixes**  $\psi :: 'v :: \text{linorder state}$   
**assumes** *resolution\*\**  $\psi\ \psi'$  **and**  $\{\#\} \in \text{fst } \psi'$   
**shows** *unsatisfiable*  $(\text{fst } \psi)$   
 $\langle \text{proof} \rangle$

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

**lemma** *simplified-falsity*:  
**assumes** *simp*: *simplified*  $\psi$   
**and**  $\{\#\} \in \psi$   
**shows**  $\psi = \{\{\#\}\}$   
 $\langle \text{proof} \rangle$



```

lemma simplify-falsity-in-preserved:
  assumes simplify  $\chi s \chi s'$ 
  and  $\{\#\} \in \chi s$ 
  shows  $\{\#\} \in \chi s'$ 
   $\langle proof \rangle$ 

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

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

lemma resolution-soundness-and-completeness':
  fixes  $\psi :: 'v :: linorder\ state$ 
  assumes
    finite: finite (fst  $\psi$ ) and
    snd: snd  $\psi = \{\}$ 
  shows  $(\exists a-u-v. (resolution^{**} \psi (\{\{\#\}\}, a-u-v))) \longleftrightarrow unsatisfiable (fst \psi)$ 
   $\langle proof \rangle$ 

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:
   $S \models_h C \longleftrightarrow \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models C$ 
   $\langle proof \rangle$ 

lemma herbrand-consistent-interp:
  consistent-interp  $(\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\})$ 
   $\langle proof \rangle$ 

lemma herbrand-total-over-set:
  total-over-set  $(\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\})\ T$ 
   $\langle proof \rangle$ 

lemma herbrand-total-over-m:
  total-over-m  $(\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\})\ T$ 
   $\langle proof \rangle$ 

```

**lemma** *herbrand-interp-iff-partial-interp-clss*:

$S \models_{hs} C \longleftrightarrow \{Pos\ P | P. P \in S\} \cup \{Neg\ P | P. P \notin S\} \models_s C$   
 $\langle proof \rangle$

**definition** *clss-lt* :: 'a::wellorder clauses  $\Rightarrow$  'a clause  $\Rightarrow$  'a clauses **where**

*clss-lt*  $N\ C = \{D \in N. D \# \subset \# C\}$

**notation** (*latex output*)

*clss-lt* ( $-\hat{<}^{bsup} - \hat{<}^{esup}$ )

**locale** *selection* =

**fixes**  $S :: 'a\ clause \Rightarrow 'a\ clause$

**assumes**

$S\text{-selects-subseteq}: \bigwedge C. S\ C \leq \# C$  **and**

$S\text{-selects-neg-lits}: \bigwedge C\ L. L \in \# S\ C \implies is\text{-neg}\ L$

**locale** *ground-resolution-with-selection* =

*selection*  $S$  **for**  $S :: ('a :: wellorder)\ clause \Rightarrow 'a\ clause$

**begin**

**context**

**fixes**  $N :: 'a\ clause\ set$

**begin**

We do not create an equivalent of  $\delta$ , but we directly defined  $N_C$  by inlining the definition.

**function**

*production* :: 'a clause  $\Rightarrow$  'a interp

**where**

*production*  $C =$

$\{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max}(\text{set-mset}\ C) = \text{Pos}\ A \wedge \text{count}\ C\ (\text{Pos}\ A) \leq 1$   
 $\wedge \neg (\bigcup D \in \{D. D \# \subset \# C\}. \text{production}\ D) \models_h C \wedge S\ C = \{\#\}\}$

$\langle proof \rangle$

**termination**  $\langle proof \rangle$

**declare** *production.simps*[*simp del*]

**definition** *interp* :: 'a clause  $\Rightarrow$  'a interp **where**

*interp*  $C = (\bigcup D \in \{D. D \# \subset \# C\}. \text{production}\ D)$

**lemma** *production-unfold*:

*production*  $C = \{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max}(\text{set-mset}\ C) = \text{Pos}\ A \wedge \text{count}\ C\ (\text{Pos}\ A) \leq 1 \wedge \neg$   
 $\text{interp}\ C \models_h C \wedge S\ C = \{\#\}\}$

$\langle proof \rangle$

**abbreviation** *productive*  $A \equiv (\text{production}\ A \neq \{\})$

**abbreviation** *produces* :: 'a clause  $\Rightarrow$  'a  $\Rightarrow$  bool **where**

*produces*  $C\ A \equiv \text{production}\ C = \{A\}$

**lemma** *producesD*:

*produces*  $C\ A \implies C \in N \wedge C \neq \{\#\} \wedge \text{Pos}\ A = \text{Max}(\text{set-mset}\ C) \wedge \text{count}\ C\ (\text{Pos}\ A) \leq 1 \wedge$   
 $\neg \text{interp}\ C \models_h C \wedge S\ C = \{\#\}$

$\langle proof \rangle$

**lemma** *produces*  $C\ A \implies \text{Pos}\ A \in \# C$

$\langle \text{proof} \rangle$

**lemma** *interp'-def-in-set:*

$\text{interp } C = (\bigcup D \in \{D \in N. D \# \subseteq \# C\}. \text{production } D)$

$\langle \text{proof} \rangle$

**lemma** *production-iff-produces:*

$\text{produces } D A \longleftrightarrow A \in \text{production } D$

$\langle \text{proof} \rangle$

**definition** *Interp* :: 'a clause  $\Rightarrow$  'a interp **where**

$\text{Interp } C = \text{interp } C \cup \text{production } C$

**lemma**

**assumes** *produces*  $C P$

**shows**  $\text{Interp } C \models_h C$

$\langle \text{proof} \rangle$

**definition** *INTERP* :: 'a interp **where**

$\text{INTERP} = (\bigcup D \in N. \text{production } D)$

**lemma** *interp-subseteq-Interp[simp]:*  $\text{interp } C \subseteq \text{Interp } C$

$\langle \text{proof} \rangle$

**lemma** *Interp-as-UNION:*  $\text{Interp } C = (\bigcup D \in \{D. D \# \subseteq \# C\}. \text{production } D)$

$\langle \text{proof} \rangle$

**lemma** *productive-not-empty:*  $\text{productive } C \Longrightarrow C \neq \{\#\}$

$\langle \text{proof} \rangle$

**lemma** *productive-imp-produces-Max-literal:*  $\text{productive } C \Longrightarrow \text{produces } C (\text{atm-of } (\text{Max } (\text{set-mset } C)))$

$\langle \text{proof} \rangle$

**lemma** *productive-imp-produces-Max-atom:*  $\text{productive } C \Longrightarrow \text{produces } C (\text{Max } (\text{atms-of } C))$

$\langle \text{proof} \rangle$

**lemma** *produces-imp-Max-literal:*  $\text{produces } C A \Longrightarrow A = \text{atm-of } (\text{Max } (\text{set-mset } C))$

$\langle \text{proof} \rangle$

**lemma** *produces-imp-Max-atom:*  $\text{produces } C A \Longrightarrow A = \text{Max } (\text{atms-of } C)$

$\langle \text{proof} \rangle$

**lemma** *produces-imp-Pos-in-lits:*  $\text{produces } C A \Longrightarrow \text{Pos } A \in \# C$

$\langle \text{proof} \rangle$

**lemma** *productive-in-N:*  $\text{productive } C \Longrightarrow C \in N$

$\langle \text{proof} \rangle$

**lemma** *produces-imp-atms-leq:*  $\text{produces } C A \Longrightarrow B \in \text{atms-of } C \Longrightarrow B \leq A$

$\langle \text{proof} \rangle$

**lemma** *produces-imp-neg-notin-lits:*  $\text{produces } C A \Longrightarrow \text{Neg } A \notin \# C$

$\langle \text{proof} \rangle$

**lemma** *less-eq-imp-interp-subseteq-interp:*  $C \# \subseteq \# D \Longrightarrow \text{interp } C \subseteq \text{interp } D$

$\langle \text{proof} \rangle$

**lemma** *less-eq-imp-interp-subseteq-Interp*:  $C \# \subseteq \# D \implies \text{interp } C \subseteq \text{Interp } D$

$\langle \text{proof} \rangle$

**lemma** *less-imp-production-subseteq-interp*:  $C \# \subset \# D \implies \text{production } C \subseteq \text{interp } D$

$\langle \text{proof} \rangle$

**lemma** *less-eq-imp-production-subseteq-Interp*:  $C \# \subseteq \# D \implies \text{production } C \subseteq \text{Interp } D$

$\langle \text{proof} \rangle$

**lemma** *less-imp-Interp-subseteq-interp*:  $C \# \subset \# D \implies \text{Interp } C \subseteq \text{interp } D$

$\langle \text{proof} \rangle$

**lemma** *less-eq-imp-Interp-subseteq-Interp*:  $C \# \subseteq \# D \implies \text{Interp } C \subseteq \text{Interp } D$

$\langle \text{proof} \rangle$

**lemma** *false-Interp-to-true-interp-imp-less-multiset*:  $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \# \subset \# D$

$\langle \text{proof} \rangle$

**lemma** *false-interp-to-true-interp-imp-less-multiset*:  $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \# \subset \# D$

$\langle \text{proof} \rangle$

**lemma** *false-Interp-to-true-Interp-imp-less-multiset*:  $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \# \subset \# D$

$\langle \text{proof} \rangle$

**lemma** *false-interp-to-true-Interp-imp-le-multiset*:  $A \notin \text{interp } C \implies A \in \text{Interp } D \implies C \# \subseteq \# D$

$\langle \text{proof} \rangle$

**lemma** *interp-subseteq-INTERP*:  $\text{interp } C \subseteq \text{INTERP}$

$\langle \text{proof} \rangle$

**lemma** *production-subseteq-INTERP*:  $\text{production } C \subseteq \text{INTERP}$

$\langle \text{proof} \rangle$

**lemma** *Interp-subseteq-INTERP*:  $\text{Interp } C \subseteq \text{INTERP}$

$\langle \text{proof} \rangle$

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

**lemma** *produces-imp-in-interp*:

**assumes** *a-in-c*:  $\text{Neg } A \in \# C$  **and** *d: produces D A*

**shows**  $A \in \text{interp } C$

$\langle \text{proof} \rangle$

**lemma** *neg-notin-Interp-not-produce*:  $\text{Neg } A \in \# C \implies A \notin \text{Interp } D \implies C \# \subseteq \# D \implies \neg \text{produces } D'' A$

$\langle \text{proof} \rangle$

**lemma** *in-production-imp-produces*:  $A \in \text{production } C \implies \text{produces } C A$

$\langle \text{proof} \rangle$

**lemma** *not-produces-imp-notin-production*:  $\neg \text{produces } C A \implies A \notin \text{production } C$

$\langle \text{proof} \rangle$

**lemma** *not-produces-imp-notin-interp*:  $(\bigwedge D. \neg \text{produces } D A) \implies A \notin \text{interp } C$

$\langle \text{proof} \rangle$

The results below corresponds to Lemma 3.4.

**Nitpicking:** If  $D = D'$  and  $D$  is productive,  $I^D \subseteq I_{D'}$  does not hold.

**lemma** *true-Interp-imp-general:*

**assumes**

*c-le-d:*  $C \# \subseteq \# D$  **and**

*d-lt-d':*  $D \# \subset \# D'$  **and**

*c-at-d:*  $\text{Interp } D \models_h C$  **and**

*subs:*  $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$

**shows**  $(\bigcup C \in CC. \text{production } C) \models_h C$

*<proof>*

**lemma** *true-Interp-imp-interp:*  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{interp } D' \models_h C$

*<proof>*

**lemma** *true-Interp-imp-Interp:*  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{Interp } D' \models_h C$

*<proof>*

**lemma** *true-Interp-imp-INTERP:*  $C \# \subseteq \# D \implies \text{Interp } D \models_h C \implies \text{INTERP} \models_h C$

*<proof>*

**lemma** *true-interp-imp-general:*

**assumes**

*c-le-d:*  $C \# \subseteq \# D$  **and**

*d-lt-d':*  $D \# \subset \# D'$  **and**

*c-at-d:*  $\text{interp } D \models_h C$  **and**

*subs:*  $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$

**shows**  $(\bigcup C \in CC. \text{production } C) \models_h C$

*<proof>*

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

**lemma** *true-interp-imp-interp:*  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{interp } D' \models_h C$

*<proof>*

**lemma** *true-interp-imp-Interp:*  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{Interp } D' \models_h C$

*<proof>*

**lemma** *true-interp-imp-INTERP:*  $C \# \subseteq \# D \implies \text{interp } D \models_h C \implies \text{INTERP} \models_h C$

*<proof>*

**lemma** *productive-imp-false-interp:*  $\text{productive } C \implies \neg \text{interp } C \models_h C$

*<proof>*

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

**lemma** *cls-gt-double-pos-no-production:*

**assumes**  $D: \{\#Pos P, Pos P\} \# \subset \# C$

**shows**  $\neg \text{produces } C P$

*<proof>*

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

**lemma**

**assumes**  $D: C + \{\#Neg P\} \# \subset \# D$

**shows**  $\text{production } D \neq \{P\}$

$\langle proof \rangle$

**lemma** *in-interp-is-produced*:

**assumes**  $P \in INTERP$

**shows**  $\exists D. D + \{\#Pos\ P\# \} \in N \wedge produces\ (D + \{\#Pos\ P\# \})\ P$

$\langle proof \rangle$

**end**

**end**

**abbreviation**  $MMax\ M \equiv Max\ (set-mset\ M)$

#### 4.2.1 We can now define the rules of the calculus

**inductive** *superposition-rules* :: '*a clause*  $\Rightarrow$  '*a clause*  $\Rightarrow$  '*a clause*  $\Rightarrow$  bool **where**

*factoring*: *superposition-rules*  $(C + \{\#Pos\ P\# \} + \{\#Pos\ P\# \})\ B\ (C + \{\#Pos\ P\# \})\ |$

*superposition-l*: *superposition-rules*  $(C_1 + \{\#Pos\ P\# \})\ (C_2 + \{\#Neg\ P\# \})\ (C_1 + C_2)$

**inductive** *superposition* :: '*a clauses*  $\Rightarrow$  '*a clauses*  $\Rightarrow$  bool **where**

*superposition*:  $A \in N \Longrightarrow B \in N \Longrightarrow superposition-rules\ A\ B\ C$

$\Longrightarrow superposition\ N\ (N \cup \{C\})$

**definition** *abstract-red* :: '*a::wellorder clause*  $\Rightarrow$  '*a clauses*  $\Rightarrow$  bool **where**

*abstract-red*  $C\ N = (clss-lt\ N\ C \models_p C)$

**lemma** *less-multiset[iff]*:  $M < N \longleftrightarrow M \# \subset \# N$

$\langle proof \rangle$

**lemma** *less-eq-multiset[iff]*:  $M \leq N \longleftrightarrow M \# \subseteq \# N$

$\langle proof \rangle$

**lemma** *herbrand-true-clss-true-clss-clss-herbrand-true-clss*:

**assumes**

$AB: A \models_{hs} B$  **and**

$BC: B \models_p C$

**shows**  $A \models_h C$

$\langle proof \rangle$

**lemma** *abstract-red-subset-mset-abstract-red*:

**assumes**

*abstr*: *abstract-red*  $C\ N$  **and**

*c-lt-d*:  $C \subseteq \# D$

**shows** *abstract-red*  $D\ N$

$\langle proof \rangle$

**lemma** *true-clss-clss-extended*:

**assumes**

$A \models_p B$  **and**

*tot*: *total-over-m*  $I\ A$  **and**

*cons*: *consistent-interp*  $I$  **and**

$I-A: I \models_s A$

**shows**  $I \models B$

$\langle proof \rangle$

**lemma**

**assumes**  
 $CP: \neg \text{class-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#C\# \} + \{ \#Neg P\# \}$  **and**  
 $\text{class-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#E\# \} + \{ \#Pos P\# \} \vee \text{class-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p$   
 $\{ \#C\# \} + \{ \#Neg P\# \}$   
**shows**  $\text{class-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#E\# \} + \{ \#Pos P\# \}$

$\langle \text{proof} \rangle$

**locale** *ground-ordered-resolution-with-redundancy* =  
*ground-resolution-with-selection* +  
**fixes** *redundant* :: 'a::wellorder clause  $\Rightarrow$  'a clauses  $\Rightarrow$  bool  
**assumes**  
*redundant-iff-abstract*:  $\text{redundant } A \ N \longleftrightarrow \text{abstract-red } A \ N$

**begin**

**definition** *saturated* :: 'a clauses  $\Rightarrow$  bool **where**

$\text{saturated } N \longleftrightarrow (\forall A \ B \ C. A \in N \longrightarrow B \in N \longrightarrow \neg \text{redundant } A \ N \longrightarrow \neg \text{redundant } B \ N$   
 $\longrightarrow \text{superposition-rules } A \ B \ C \longrightarrow \text{redundant } C \ N \vee C \in N)$

**lemma**

**assumes**  
*saturated*: *saturated* *N* **and**  
*finite*: *finite* *N* **and**  
*empty*:  $\{ \# \} \notin N$   
**shows** *INTERP* *N*  $\models_{hs}$  *N*

$\langle \text{proof} \rangle$

**end**

**lemma** *tautology-is-redundant*:

**assumes** *tautology* *C*  
**shows** *abstract-red* *C* *N*  
 $\langle \text{proof} \rangle$

**lemma** *subsumed-is-redundant*:

**assumes** *AB*:  $A \subset \# B$   
**and** *AN*:  $A \in N$   
**shows** *abstract-red* *B* *N*  
 $\langle \text{proof} \rangle$

**inductive** *redundant* :: 'a clause  $\Rightarrow$  'a clauses  $\Rightarrow$  bool **where**

*subsumption*:  $A \in N \Longrightarrow A \subset \# B \Longrightarrow \text{redundant } B \ N$

**lemma** *redundant-is-redundancy-criterion*:

**fixes** *A* :: 'a :: wellorder clause **and** *N* :: 'a :: wellorder clauses  
**assumes** *redundant* *A* *N*  
**shows** *abstract-red* *A* *N*  
 $\langle \text{proof} \rangle$

**lemma** *redundant-mono*:

$\text{redundant } A \ N \Longrightarrow A \subseteq \# B \Longrightarrow \text{redundant } B \ N$   
 $\langle \text{proof} \rangle$

**locale** *truc* =

*selection* *S* **for** *S* :: nat clause  $\Rightarrow$  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**  
   $\bigwedge L \ xs. P \ xs \implies P \ (\text{Decided } L \ \# \ xs)$  **and**  
   $\bigwedge L \ m \ xs. P \ xs \implies P \ (\text{Propagated } L \ m \ \# \ xs)$   
  **shows**  $P \ xs$   
   $\langle \text{proof} \rangle$

**lemma** *is-decided-ex-Decided*:  
   $\text{is-decided } L \implies (\bigwedge K. L = \text{Decided } K \implies P) \implies P$   
   $\langle \text{proof} \rangle$

**type-synonym** ('v, 'm) *ann-lits* = ('v, 'm) *ann-lit list*

**definition** *lits-of* :: ('a, 'b) *ann-lit set*  $\Rightarrow$  'a *literal set* **where**  
*lits-of* *Ls* = *lit-of* ' *Ls*

**abbreviation** *lits-of-l* :: ('a, 'b) *ann-lits*  $\Rightarrow$  'a *literal set* **where**  
*lits-of-l* *Ls*  $\equiv$  *lits-of* (set *Ls*)

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

**lemma** *lits-of-insert*[simp]:  
  *lits-of* (insert *L* *Ls*) = insert (lit-of *L*) (*lits-of* *Ls*)  
   $\langle \text{proof} \rangle$

**lemma** *lits-of-l-Un*[simp]:  
  *lits-of* ( $l \cup l'$ ) = *lits-of* *l*  $\cup$  *lits-of* *l'*  
   $\langle \text{proof} \rangle$

**lemma** *finite-lits-of-def*[simp]:  
  finite (*lits-of-l* *L*)  
   $\langle \text{proof} \rangle$



**abbreviation** *unmark* **where**  
 $unmark \equiv (\lambda a. \{\#lit\text{-of } a\# \})$

**abbreviation** *unmark-s* **where**  
 $unmark\text{-}s\ M \equiv unmark\ 'M$

**abbreviation** *unmark-l* **where**  
 $unmark\text{-}l\ M \equiv unmark\text{-}s\ (set\ M)$

**lemma** *atms-of-ms-lambda-lit-of-is-atm-of-lit-of[simp]*:  
 $atms\text{-of}\text{-}ms\ (unmark\text{-}l\ M') = atm\text{-of}\ ' lits\text{-of}\text{-}l\ M'$   
 $\langle proof \rangle$

**lemma** *lits-of-l-empty-is-empty[iff]*:  
 $lits\text{-of}\text{-}l\ M = \{\} \longleftrightarrow M = []$   
 $\langle proof \rangle$

## Entailment

**definition** *true-annot* :: (*'a*, *'m*) *ann-lits*  $\Rightarrow$  *'a clause*  $\Rightarrow$  *bool* (**infix**  $\models_a$  49) **where**  
 $I \models_a C \longleftrightarrow (lits\text{-of}\text{-}l\ I) \models C$

**definition** *true-annots* :: (*'a*, *'m*) *ann-lits*  $\Rightarrow$  *'a clauses*  $\Rightarrow$  *bool* (**infix**  $\models_{as}$  49) **where**  
 $I \models_{as} CC \longleftrightarrow (\forall C \in CC. I \models_a C)$

**lemma** *true-annot-empty-model[simp]*:  
 $\neg[] \models_a \psi$   
 $\langle proof \rangle$

**lemma** *true-annot-empty[simp]*:  
 $\neg I \models_a \{\#\}$   
 $\langle proof \rangle$

**lemma** *empty-true-annots-def[iff]*:  
 $[] \models_{as} \psi \longleftrightarrow \psi = \{\}$   
 $\langle proof \rangle$

**lemma** *true-annots-empty[simp]*:  
 $I \models_{as} \{\}$   
 $\langle proof \rangle$

**lemma** *true-annots-single-true-annot[iff]*:  
 $I \models_{as} \{C\} \longleftrightarrow I \models_a C$   
 $\langle proof \rangle$

**lemma** *true-annot-insert-l[simp]*:  
 $M \models_a A \Longrightarrow L \# M \models_a A$   
 $\langle proof \rangle$

**lemma** *true-annots-insert-l [simp]*:  
 $M \models_{as} A \Longrightarrow L \# M \models_{as} A$   
 $\langle proof \rangle$

**lemma** *true-annots-union[iff]*:  
 $M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$

$\langle proof \rangle$

**lemma** *true-annots-insert*[*iff*]:

$M \models_{as} insert\ a\ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$

$\langle proof \rangle$

Link between  $\models_{as}$  and  $\models_s$ :

**lemma** *true-annots-true-clss*:

$I \models_{as} CC \longleftrightarrow lits-of-l\ I \models_s CC$

$\langle proof \rangle$

**lemma** *in-lit-of-true-annot*:

$a \in lits-of-l\ M \longleftrightarrow M \models_a \{\#a\#\}$

$\langle proof \rangle$

**lemma** *true-annot-lit-of-notin-skip*:

$L \# M \models_a A \implies lit-of\ L \notin \# A \implies M \models_a A$

$\langle proof \rangle$

**lemma** *true-clss-singleton-lit-of-implies-incl*:

$I \models_s unmark-l\ MLs \implies lits-of-l\ MLs \subseteq I$

$\langle proof \rangle$

**lemma** *true-annot-true-clss-clss*:

$MLs \models_a \psi \implies set\ (map\ unmark\ MLs) \models_p \psi$

$\langle proof \rangle$

**lemma** *true-annots-true-clss-clss*:

$MLs \models_{as} \psi \implies set\ (map\ unmark\ MLs) \models_{ps} \psi$

$\langle proof \rangle$

**lemma** *true-annots-decided-true-clss*[*iff*]:

$map\ Decided\ M \models_{as} N \longleftrightarrow set\ M \models_s N$

$\langle proof \rangle$

**lemma** *true-annot-singleton*[*iff*]:  $M \models_a \{\#L\#\} \longleftrightarrow L \in lits-of-l\ M$

$\langle proof \rangle$

**lemma** *true-annots-true-clss-clss*:

$A \models_{as} \Psi \implies unmark-l\ A \models_{ps} \Psi$

$\langle proof \rangle$

**lemma** *true-annot-commute*:

$M @ M' \models_a D \longleftrightarrow M' @ M \models_a D$

$\langle proof \rangle$

**lemma** *true-annots-commute*:

$M @ M' \models_{as} D \longleftrightarrow M' @ M \models_{as} D$

$\langle proof \rangle$

**lemma** *true-annot-mono*[*dest*]:

$set\ I \subseteq set\ I' \implies I \models_a N \implies I' \models_a N$

$\langle proof \rangle$

**lemma** *true-annots-mono*:

$set\ I \subseteq set\ I' \implies I \models_{as} N \implies I' \models_{as} N$   
 $\langle proof \rangle$

## Defined and undefined literals

We introduce the functions *defined-lit* and *undefined-lit* to know whether a literal is defined with respect to a list of decided literals (aka a trail in most cases).

Remark that *undefined* already exists and is a completely different Isabelle function.

**definition** *defined-lit* :: ('a, 'm) ann-lits  $\Rightarrow$  'a literal  $\Rightarrow$  bool

**where**

*defined-lit* I L  $\longleftrightarrow$  (Decided L  $\in$  set I)  $\vee$  ( $\exists$  P. Propagated L P  $\in$  set I)  
 $\vee$  (Decided ( $-$ L)  $\in$  set I)  $\vee$  ( $\exists$  P. Propagated ( $-$ L) P  $\in$  set I)

**abbreviation** *undefined-lit* :: ('a, 'm) ann-lits  $\Rightarrow$  'a literal  $\Rightarrow$  bool

**where** *undefined-lit* I L  $\equiv \neg$ *defined-lit* I L

**lemma** *defined-lit-rev[simp]*:

*defined-lit* (rev M) L  $\longleftrightarrow$  *defined-lit* M L

$\langle proof \rangle$

**lemma** *atm-imp-decided-or-proped*:

**assumes** x  $\in$  set I

**shows**

(Decided ( $-$  lit-of x)  $\in$  set I)  
 $\vee$  (Decided (lit-of x)  $\in$  set I)  
 $\vee$  ( $\exists$  l. Propagated ( $-$  lit-of x) l  $\in$  set I)  
 $\vee$  ( $\exists$  l. Propagated (lit-of x) l  $\in$  set I)

$\langle proof \rangle$

**lemma** *literal-is-lit-of-decided*:

**assumes** L = lit-of x

**shows** (x = Decided L)  $\vee$  ( $\exists$  l'. x = Propagated L l')

$\langle proof \rangle$

**lemma** *true-annot-iff-decided-or-true-lit*:

*defined-lit* I L  $\longleftrightarrow$  (lits-of-l I  $\models$  L  $\vee$  lits-of-l I  $\models$   $-$ L)

$\langle proof \rangle$

**lemma** *consistent-inter-true-annots-satisfiable*:

*consistent-interp* (lits-of-l I)  $\implies$  I  $\models_{as}$  N  $\implies$  satisfiable N

$\langle proof \rangle$

**lemma** *defined-lit-map*:

*defined-lit* Ls L  $\longleftrightarrow$  atm-of L  $\in$  ( $\lambda$ l. atm-of (lit-of l)) ' set Ls

$\langle proof \rangle$

**lemma** *defined-lit-uminus[iff]*:

*defined-lit* I ( $-$ L)  $\longleftrightarrow$  *defined-lit* I L

$\langle proof \rangle$

**lemma** *Decided-Propagated-in-iff-in-lits-of-l*:

*defined-lit* I L  $\longleftrightarrow$  (L  $\in$  lits-of-l I  $\vee$   $-$ L  $\in$  lits-of-l I)

$\langle proof \rangle$

**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)*)  
 $\langle \text{proof} \rangle$

**lemma** *decided-empty[simp]*:  
 $\neg \text{defined-lit } [] \ L$   
 $\langle \text{proof} \rangle$

### 4.3.2 Backtracking

**fun** *backtrack-split* :: (*'v*, *'m*) *ann-lits*  
 $\Rightarrow$  (*'v*, *'m*) *ann-lits*  $\times$  (*'v*, *'m*) *ann-lits* **where**  
*backtrack-split* [] = ([], []) |  
*backtrack-split* (*Propagated L P # mlits*) = *apfst* ((*op #*) (*Propagated L P*)) (*backtrack-split mlits*) |  
*backtrack-split* (*Decided L # mlits*) = ([], *Decided L # mlits*)

**lemma** *backtrack-split-fst-not-decided*:  $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \Rightarrow \neg \text{is-decided } a$   
 $\langle \text{proof} \rangle$

**lemma** *backtrack-split-snd-hd-decided*:  
 $\text{snd } (\text{backtrack-split } l) \neq [] \Rightarrow \text{is-decided } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$   
 $\langle \text{proof} \rangle$

**lemma** *backtrack-split-list-eq[simp]*:  
 $\text{fst } (\text{backtrack-split } l) @ (\text{snd } (\text{backtrack-split } l)) = l$   
 $\langle \text{proof} \rangle$

**lemma** *backtrack-snd-empty-not-decided*:  
 $\text{backtrack-split } M = (M'', []) \Rightarrow \forall l \in \text{set } M. \neg \text{is-decided } l$   
 $\langle \text{proof} \rangle$

**lemma** *backtrack-split-some-is-decided-then-snd-has-hd*:  
 $\exists l \in \text{set } M. \text{is-decided } l \Rightarrow \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$   
 $\langle \text{proof} \rangle$

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*:  
 $\text{backtrack-split } M = (\text{takeWhile } (\text{Not } o \text{ is-decided}) \ M, \text{dropWhile } (\text{Not } o \text{ is-decided}) \ M)$   
 $\langle \text{proof} \rangle$

### 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*  
 $\Rightarrow ((\text{'a}, \text{'m}) \text{ ann-lits} \times (\text{'a}, \text{'m}) \text{ ann-lits}) \text{ list}$  **where**

```

get-all-ann-decomposition (Decided L # Ls) =
  (Decided L # Ls, []) # get-all-ann-decomposition Ls |
get-all-ann-decomposition (Propagated L P # Ls) =
  (apsnd ((op #) (Propagated L P)) (hd (get-all-ann-decomposition Ls)))
  # tl (get-all-ann-decomposition Ls) |
get-all-ann-decomposition [] = [([], [])]

value get-all-ann-decomposition [Propagated A5 B5, Decided C4, Propagated A3 B3,
  Propagated A2 B2, Decided C1, Propagated A0 B0]

```

Now we can prove several simple properties about the function.

**lemma** *get-all-ann-decomposition-never-empty*[iff]:  
*get-all-ann-decomposition M = []  $\longleftrightarrow$  False*  
*<proof>*

**lemma** *get-all-ann-decomposition-never-empty-sym*[iff]:  
*[] = get-all-ann-decomposition M  $\longleftrightarrow$  False*  
*<proof>*

**lemma** *get-all-ann-decomposition-decomp*:  
*hd (get-all-ann-decomposition S) = (a, c)  $\implies$  S = c @ a*  
*<proof>*

**lemma** *get-all-ann-decomposition-backtrack-split*:  
*backtrack-split S = (M, M')  $\longleftrightarrow$  hd (get-all-ann-decomposition S) = (M', M)*  
*<proof>*

**lemma** *get-all-ann-decomposition-Nil-backtrack-split-snd-Nil*:  
*get-all-ann-decomposition S = [([], A)]  $\implies$  snd (backtrack-split S) = []*  
*<proof>*

This functions says that the first element is either empty or starts with a decided element of the list.

**lemma** *get-all-ann-decomposition-length-1-fst-empty-or-length-1*:  
**assumes** *get-all-ann-decomposition M = (a, b) # []*  
**shows** *a = []  $\vee$  (length a = 1  $\wedge$  is-decided (hd a)  $\wedge$  hd a  $\in$  set M)*  
*<proof>*

**lemma** *get-all-ann-decomposition-fst-empty-or-hd-in-M*:  
**assumes** *get-all-ann-decomposition M = (a, b) # l*  
**shows** *a = []  $\vee$  (is-decided (hd a)  $\wedge$  hd a  $\in$  set M)*  
*<proof>*

**lemma** *get-all-ann-decomposition-snd-not-decided*:  
**assumes** *(a, b)  $\in$  set (get-all-ann-decomposition M)*  
**and** *L  $\in$  set b*  
**shows**  *$\neg$ is-decided L*  
*<proof>*

**lemma** *tl-get-all-ann-decomposition-skip-some*:  
**assumes** *x  $\in$  set (tl (get-all-ann-decomposition M1))*  
**shows** *x  $\in$  set (tl (get-all-ann-decomposition (M0 @ M1)))*  
*<proof>*

**lemma** *hd-get-all-ann-decomposition-skip-some*:

**assumes**  $(x, y) = \text{hd } (\text{get-all-ann-decomposition } M1)$   
**shows**  $(x, y) \in \text{set } (\text{get-all-ann-decomposition } (M0 @ \text{Decided } K \# M1))$   
 $\langle \text{proof} \rangle$

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

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

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

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

**lemma** *get-all-ann-decomposition-last-choice:*  
**assumes**  $\text{tl } (\text{get-all-ann-decomposition } (M' @ \text{Decided } L \# M)) \neq []$   
**and**  $\forall l \in \text{set } M'. \neg \text{is-decided } l$   
**and**  $\text{hd } (\text{tl } (\text{get-all-ann-decomposition } (M' @ \text{Decided } L \# M))) = (M0', M0)$   
**shows**  $\text{hd } (\text{get-all-ann-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *get-all-ann-decomposition-incl:*  
**assumes**  $(a, b) \in \text{set } (\text{get-all-ann-decomposition } M)$   
**shows**  $\text{set } b \subseteq \text{set } M$  **and**  $\text{set } a \subseteq \text{set } M$   
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *union-in-get-all-ann-decomposition-is-subset:*

**assumes**  $(a, b) \in \text{set } (\text{get-all-ann-decomposition } M)$

**shows**  $\text{set } a \cup \text{set } b \subseteq \text{set } M$

$\langle \text{proof} \rangle$

**lemma** *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons:*

$\exists M1\ M2. (\text{Decided } K \# M1, M2) \in \text{set } (\text{get-all-ann-decomposition } (c @ \text{Decided } K \# c'))$

$\langle \text{proof} \rangle$

**lemma** *fst-get-all-ann-decomposition-prepend-not-decided:*

**assumes**  $\forall m \in \text{set } MS. \neg \text{is-decided } m$

**shows**  $\text{set } (\text{map } \text{fst } (\text{get-all-ann-decomposition } M))$   
 $= \text{set } (\text{map } \text{fst } (\text{get-all-ann-decomposition } (MS @ M)))$

$\langle \text{proof} \rangle$

## Entailment of the Propagated by the Decided Literal

**lemma** *get-all-ann-decomposition-snd-union:*

$\text{set } M = \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-ann-decomposition } M)) \cup \{L \mid L. \text{is-decided } L \wedge L \in \text{set } M\}$   
(**is**  $?M\ M = ?U\ M \cup ?Ls\ M$ )

$\langle \text{proof} \rangle$

**definition** *all-decomposition-implies :: 'a literal multiset set*

$\Rightarrow ((('a, 'm) \text{ ann-lits} \times ('a, 'm) \text{ ann-lits}) \text{ list} \Rightarrow \text{bool})$  **where**

*all-decomposition-implies*  $N\ S \longleftrightarrow (\forall (Ls, \text{seen}) \in \text{set } S. \text{unmark-l } Ls \cup N \models_{ps} \text{unmark-l } \text{seen})$

**lemma** *all-decomposition-implies-empty[iff]:*

*all-decomposition-implies*  $N\ [] \langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-single[iff]:*

*all-decomposition-implies*  $N\ [(Ls, \text{seen})] \longleftrightarrow \text{unmark-l } Ls \cup N \models_{ps} \text{unmark-l } \text{seen}$

$\langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-append[iff]:*

*all-decomposition-implies*  $N\ (S @ S')$

$\longleftrightarrow (\text{all-decomposition-implies } N\ S \wedge \text{all-decomposition-implies } N\ S')$

$\langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-cons-pair[iff]:*

*all-decomposition-implies*  $N\ ((Ls, \text{seen}) \# S')$

$\longleftrightarrow (\text{all-decomposition-implies } N\ [(Ls, \text{seen})] \wedge \text{all-decomposition-implies } N\ S')$

$\langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-cons-single[iff]:*

*all-decomposition-implies*  $N\ (l \# S') \longleftrightarrow$

$(\text{unmark-l } (\text{fst } l) \cup N \models_{ps} \text{unmark-l } (\text{snd } l) \wedge$

$\text{all-decomposition-implies } N\ S')$

$\langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-trail-is-implied:*

**assumes** *all-decomposition-implies*  $N\ (\text{get-all-ann-decomposition } M)$

**shows**  $N \cup \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\}$

$\models_{ps} \text{unmark } ' \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-ann-decomposition } M))$

$\langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-propagated-lits-are-implied*:  
**assumes** *all-decomposition-implies*  $N$  (*get-all-ann-decomposition*  $M$ )  
**shows**  $N \cup \{\text{unmark } L \mid L. \text{ is-decided } L \wedge L \in \text{set } M\} \models_{ps} \text{unmark-l } M$   
 (**is**  $?I \models_{ps} ?A$ )  
 $\langle \text{proof} \rangle$

**lemma** *all-decomposition-implies-insert-single*:  
*all-decomposition-implies*  $N \ M \implies \text{all-decomposition-implies } (\text{insert } C \ N) \ M$   
 $\langle \text{proof} \rangle$

#### 4.3.4 Negation of Clauses

We define the negation of a '*a Partial-Clausal-Logic.clause*': it converts it from the a single clause to a set of clauses, wherein each clause is a single negated literal.

**definition** *CNot* :: '*v clause*  $\Rightarrow$  '*v clauses* **where**  
 $CNot \ \psi = \{ \{ \# - L \# \} \mid L. L \in \# \ \psi \}$

**lemma** *in-CNot-uminus[iff]*:  
**shows**  $\{ \# L \# \} \in CNot \ \psi \longleftrightarrow -L \in \# \ \psi$   
 $\langle \text{proof} \rangle$

**lemma**  
**shows**  
*CNot-singleton[simp]*:  $CNot \ \{ \# L \# \} = \{ \{ \# - L \# \} \}$  **and**  
*CNot-empty[simp]*:  $CNot \ \{ \# \} = \{ \}$  **and**  
*CNot-plus[simp]*:  $CNot \ (A + B) = CNot \ A \cup CNot \ B$   
 $\langle \text{proof} \rangle$

**lemma** *CNot-eq-empty[iff]*:  
 $CNot \ D = \{ \} \longleftrightarrow D = \{ \# \}$   
 $\langle \text{proof} \rangle$

**lemma** *in-CNot-implies-uminus*:  
**assumes**  $L \in \# \ D$  **and**  $M \models_{as} CNot \ D$   
**shows**  $M \models_a \{ \# - L \# \}$  **and**  $-L \in \text{lits-of-l } M$   
 $\langle \text{proof} \rangle$

**lemma** *CNot-remdups-mset[simp]*:  
 $CNot \ (\text{remdups-mset } A) = CNot \ A$   
 $\langle \text{proof} \rangle$

**lemma** *Ball-CNot-Ball-mset[simp]*:  
 $(\forall x \in CNot \ D. P \ x) \longleftrightarrow (\forall L \in \# \ D. P \ \{ \# - L \# \})$   
 $\langle \text{proof} \rangle$

**lemma** *consistent-CNot-not*:  
**assumes** *consistent-interp*  $I$   
**shows**  $I \models_s CNot \ \varphi \implies \neg I \models \varphi$   
 $\langle \text{proof} \rangle$

**lemma** *total-not-true-cls-true-clss-CNot*:  
**assumes** *total-over-m*  $I \ \{ \varphi \}$  **and**  $\neg I \models \varphi$   
**shows**  $I \models_s CNot \ \varphi$   
 $\langle \text{proof} \rangle$



**lemma** *total-not-CNot*:

**assumes** *total-over-m*  $I \{ \varphi \}$  **and**  $\neg I \models_s CNot \varphi$   
**shows**  $I \models \varphi$   
 $\langle proof \rangle$

**lemma** *atms-of-ms-CNot-atms-of[simp]*:

*atms-of-ms* ( $CNot C$ ) = *atms-of*  $C$   
 $\langle proof \rangle$

**lemma** *true-clss-clss-contradiction-true-clss-clss-false*:

$C \in D \implies D \models_{ps} CNot C \implies D \models_p \{ \# \}$   
 $\langle proof \rangle$

**lemma** *true-annots-CNot-all-atms-defined*:

**assumes**  $M \models_{as} CNot T$  **and**  $a1: L \in \# T$   
**shows**  $atm-of L \in atm-of \text{' lits-of-l } M$   
 $\langle proof \rangle$

**lemma** *true-annots-CNot-all-uminus-atms-defined*:

**assumes**  $M \models_{as} CNot T$  **and**  $a1: -L \in \# T$   
**shows**  $atm-of L \in atm-of \text{' lits-of-l } M$   
 $\langle proof \rangle$

**lemma** *true-clss-clss-false-left-right*:

**assumes**  $\{ \{ \#L\# \} \} \cup B \models_p \{ \# \}$   
**shows**  $B \models_{ps} CNot \{ \#L\# \}$   
 $\langle proof \rangle$

**lemma** *true-annots-true-clss-def-iff-negation-in-model*:

$M \models_{as} CNot C \longleftrightarrow (\forall L \in \# C. -L \in lits-of-l M)$   
 $\langle proof \rangle$

**lemma** *true-annot-CNot-diff*:

$I \models_{as} CNot C \implies I \models_{as} CNot (C - C')$   
 $\langle proof \rangle$

**lemma** *CNot-mset-replicate[simp]*:

$CNot (mset (replicate n L)) = (if n = 0 then \{ \} else \{ \{ \# - L\# \} \})$   
 $\langle proof \rangle$

**lemma** *consistent-CNot-not-tautology*:

*consistent-interp*  $M \implies M \models_s CNot D \implies \neg tautology D$   
 $\langle proof \rangle$

**lemma** *atms-of-ms-CNot-atms-of-ms*: *atms-of-ms* ( $CNot CC$ ) = *atms-of-ms*  $\{ CC \}$

$\langle proof \rangle$

**lemma** *total-over-m-CNot-toal-over-m[simp]*:

*total-over-m*  $I (CNot C) = total-over-set I (atms-of C)$   
 $\langle proof \rangle$

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*:  $-(a::'a \text{ literal}) = i \longleftrightarrow a = -i$

$\langle \text{proof} \rangle$

**lemma** *true-clss-clss-plus-CNot*:

**assumes**

$CC-L: A \models_p CC + \{\#L\# \}$  **and**

$CNot-CC: A \models_{ps} CNot\ CC$

**shows**  $A \models_p \{\#L\# \}$

$\langle \text{proof} \rangle$

**lemma** *true-annots-CNot-lit-of-notin-skip*:

**assumes**  $LM: L \# M \models_{as} CNot\ A$  **and**  $LA: \text{lit-of } L \notin \# A \rightarrow \text{lit-of } L \notin \# A$

**shows**  $M \models_{as} CNot\ A$

$\langle \text{proof} \rangle$

**lemma** *true-clss-clss-union-false-true-clss-clss-cnot*:

$A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} CNot\ B$

$\langle \text{proof} \rangle$

**lemma** *true-annot-remove-hd-if-notin-vars*:

**assumes**  $a \# M' \models_a D$  **and**  $\text{atm-of } (\text{lit-of } a) \notin \text{atms-of } D$

**shows**  $M' \models_a D$

$\langle \text{proof} \rangle$

**lemma** *true-annot-remove-if-notin-vars*:

**assumes**  $M @ M' \models_a D$  **and**  $\forall x \in \text{atms-of } D. x \notin \text{atm-of } \text{'lits-of-l } M$

**shows**  $M' \models_a D$

$\langle \text{proof} \rangle$

**lemma** *true-annots-remove-if-notin-vars*:

**assumes**  $M @ M' \models_{as} D$  **and**  $\forall x \in \text{atms-of-ms } D. x \notin \text{atm-of } \text{'lits-of-l } M$

**shows**  $M' \models_{as} D$   $\langle \text{proof} \rangle$

**lemma** *all-variables-defined-not-imply-cnot*:

**assumes**

$\forall s \in \text{atms-of-ms } \{B\}. s \in \text{atm-of } \text{'lits-of-l } A$  **and**

$\neg A \models_a B$

**shows**  $A \models_{as} CNot\ B$

$\langle \text{proof} \rangle$

**lemma** *CNot-union-mset[simp]*:

$CNot\ (A \# \cup B) = CNot\ A \cup CNot\ B$

$\langle \text{proof} \rangle$

### 4.3.5 Other

**abbreviation**  $\text{no-dup } L \equiv \text{distinct } (\text{map } (\lambda l. \text{atm-of } (\text{lit-of } l))\ L)$

**lemma** *no-dup-rev[simp]*:

$\text{no-dup } (\text{rev } M) \longleftrightarrow \text{no-dup } M$

$\langle \text{proof} \rangle$

**lemma** *no-dup-length-eq-card-atm-of-lits-of-l*:

**assumes**  $\text{no-dup } M$

**shows**  $\text{length } M = \text{card } (\text{atm-of } \text{'lits-of-l } M)$

$\langle \text{proof} \rangle$

**lemma** *distinct-consistent-interp*:  
 $no\_dup\ M \implies consistent\_interp\ (lits\_of\_l\ M)$   
 $\langle proof \rangle$

**lemma** *distinct-get-all-ann-decomposition-no-dup*:  
**assumes**  $(a, b) \in set\ (get\_all\_ann\_decomposition\ M)$   
**and**  $no\_dup\ M$   
**shows**  $no\_dup\ (a\ @\ b)$   
 $\langle proof \rangle$

**lemma** *true-annots-lit-of-notin-skip*:  
**assumes**  $L\ \# \ M \models_{as} CNot\ A$   
**and**  $\neg lit\_of\ L \notin \# \ A$   
**and**  $no\_dup\ (L\ \# \ M)$   
**shows**  $M \models_{as} CNot\ A$   
 $\langle proof \rangle$

#### 4.3.6 Extending Entailments to multisets

We have defined previous entailment with respect to sets, but we also need a multiset version depending on the context. The conversion is simple using the function *set-mset* (in this direction, there is no loss of information).

**abbreviation** *true-annots-mset* (**infix**  $\models_{asm}$  50) **where**  
 $I \models_{asm} C \equiv I \models_{as} (set\_mset\ C)$

**abbreviation** *true-clss-clss-m*::  $'v\ clause\ multiset \Rightarrow 'v\ clause\ multiset \Rightarrow bool$  (**infix**  $\models_{psm}$  50)  
**where**  
 $I \models_{psm} C \equiv set\_mset\ I \models_{ps} (set\_mset\ C)$

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

**lemma** *true-clss-clssm-subsetE*:  $N \models_{psm} B \implies A \subseteq \# \ B \implies N \models_{psm} A$   
 $\langle proof \rangle$

**abbreviation** *true-clss-clss-m*::  $'a\ clause\ multiset \Rightarrow 'a\ clause \Rightarrow bool$  (**infix**  $\models_{pm}$  50) **where**  
 $I \models_{pm} C \equiv set\_mset\ I \models_p C$

**abbreviation** *distinct-mset-mset* ::  $'a\ multiset\ multiset \Rightarrow bool$  **where**  
 $distinct\_mset\_mset\ \Sigma \equiv distinct\_mset\_set\ (set\_mset\ \Sigma)$

**abbreviation** *all-decomposition-implies-m* **where**  
 $all\_decomposition\_implies\_m\ A\ B \equiv all\_decomposition\_implies\ (set\_mset\ A)\ B$

**abbreviation** *atms-of-mm* ::  $'a\ literal\ multiset\ multiset \Rightarrow 'a\ set$  **where**  
 $atms\_of\_mm\ U \equiv atms\_of\_ms\ (set\_mset\ U)$

Other definition using *Union-mset*

**lemma** *atms-of-mm*  $U \equiv set\_mset\ (\bigcup \# \ image\_mset\ (image\_mset\ atm\_of)\ U)$   
 $\langle proof \rangle$

**abbreviation** *true-clss-m*::  $'a\ interp \Rightarrow 'a\ clause\ multiset \Rightarrow bool$  (**infix**  $\models_{sm}$  50) **where**  
 $I \models_{sm} C \equiv I \models_s set\_mset\ C$

**abbreviation** *true-clss-ext-m* (**infix**  $\models_{sextm}$  49) **where**  
 $I \models_{sextm} C \equiv I \models_{sext} set\_mset\ C$

```
type-synonym 'v clauses = 'v clause multiset
end
```

## Chapter 5

# NOT's CDCL and DPLL

```
theory CDCL-WNOT-Measure
imports Main List-More
begin
```

The organisation of the development is the following:

- `CDCL_WNOT_Measure.thy` contains the measure used to show the termination the core of CDCL.
- `CDCL_NOT.thy` contains the specification of the rules: the rules are defined, and we proof the correctness and termination for some strategies CDCL.
- `DPLL_NOT.thy` contains the DPLL calculus based on the CDCL version.
- `DPLL_W.thy` contains Weidenbach's version of DPLL and the proof of equivalence between the two DPLL versions.

### 5.1 Measure

This measure show the termination of the core of CDCL: each step improves the number of literals we know for sure.

This measure can also be seen as the increasing lexicographic order: it is an order on bounded sequences, when each element is bounded. The proof involves a measure like the one defined here (the same?).

**definition**  $\mu_C :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat}$  **where**  
 $\mu_C s b M \equiv (\sum i=0..<\text{length } M. M!i * b^{\wedge} (s + i - \text{length } M))$

**lemma**  $\mu_C\text{-Nil}[simp]$ :  
 $\mu_C s b [] = 0$   
*<proof>*

**lemma**  $\mu_C\text{-single}[simp]$ :  
 $\mu_C s b [L] = L * b^{\wedge} (s - \text{Suc } 0)$   
*<proof>*

**lemma**  $\text{set-sum-atLeastLessThan-add}$ :  
 $(\sum i=k..<k+(b::\text{nat}). f i) = (\sum i=0..<b. f (k + i))$   
*<proof>*

**lemma** *set-sum-atLeastLessThan-Suc*:

$(\sum_{i=1..<Suc\ j}. f\ i) = (\sum_{i=0..<j}. f\ (Suc\ i))$   
 $\langle proof \rangle$

**lemma**  $\mu_C$ -cons:

$\mu_C\ s\ b\ (L\ \# \ M) = L * b^{\wedge (s - 1 - length\ M)} + \mu_C\ s\ b\ M$   
 $\langle proof \rangle$

**lemma**  $\mu_C$ -append:

**assumes**  $s \geq length\ (M@M')$   
**shows**  $\mu_C\ s\ b\ (M@M') = \mu_C\ (s - length\ M')\ b\ M + \mu_C\ s\ b\ M'$   
 $\langle proof \rangle$

**lemma**  $\mu_C$ -cons-non-empty-inf:

**assumes**  $M$ -ge-1:  $\forall i \in set\ M. i \geq 1$  **and**  $M: M \neq []$   
**shows**  $\mu_C\ s\ b\ M \geq b^{\wedge (s - length\ M)}$   
 $\langle proof \rangle$

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

**lemma** *sum-of-powers*:  $0 \leq k \implies (k - 1) * (\sum_{i=0..<n}. k^i) = k^n - (1::nat)$   
 $\langle proof \rangle$

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

**lemma**  $\mu_C$ -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**  $\mu_C\ s\ b\ M < b^{\wedge s}$   
 $\langle proof \rangle$

In the degenerate case  $b = (0::'a)$ , the list  $M$  is empty (since the list cannot contain any element).

**lemma**  $\mu_C$ -bounded:

**fixes**  $b :: nat$   
**assumes**  
 $M$ -le:  $\forall i < length\ M. M!i < b$  **and**  
 $s \geq length\ M$   
 $b > 0$   
**shows**  $\mu_C\ s\ b\ M < b^{\wedge s}$   
 $\langle proof \rangle$

When  $b = 0$ , we cannot show that the measure is empty, since  $0^0 = 1$ .

**lemma**  $\mu_C$ -base-0:

**assumes**  $length\ M \leq s$   
**shows**  $\mu_C\ s\ 0\ M \leq M!0$   
 $\langle proof \rangle$

**lemma** *finite-bounded-pair-list*:

**fixes**  $b :: nat$   
**shows**  $finite\ \{(ys, xs). length\ xs < s \wedge length\ ys < s \wedge$

$(\forall i < \text{length } xs. xs ! i < b) \wedge (\forall i < \text{length } ys. ys ! i < b)\}$   
 $\langle \text{proof} \rangle$

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

**lemma**  $\text{finite-}\nu NOT[simp]$ :  
 $\text{finite } (\nu NOT\ s\ base)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{acyclic-}\nu NOT$ :  $\text{acyclic } (\nu NOT\ s\ base)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{wf-}\nu NOT$ :  $\text{wf } (\nu NOT\ s\ base)$   
 $\langle \text{proof} \rangle$

**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**  $\text{no-dup-cannot-not-lit-and-uminus}$ :  
 $\text{no-dup } M \Longrightarrow -\text{lit-of } xa = \text{lit-of } x \Longrightarrow x \in \text{set } M \Longrightarrow xa \notin \text{set } M$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{atms-of-ms-single-atm-of}[simp]$ :  
 $\text{atms-of-ms } \{\text{unmark } L \mid L. P\ L\} = \text{atm-of } ' \{\text{lit-of } L \mid L. P\ L\}$   
 $\langle \text{proof} \rangle$

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

**lemma**  $\text{atms-of-ms-single-image-atm-of-lit-of}$ :  
 $\text{atms-of-ms } (\text{unmark-s } A) = \text{atm-of } ' (\text{lit-of } ' A)$   
 $\langle \text{proof} \rangle$

### 5.2.2 Initial definitions

**The state**

We define here an abstraction over operation on the state we are manipulating.

**locale**  $\text{dpll-state-ops} =$   
**fixes**  
 $\text{trail} :: 'st \Rightarrow ('v, \text{unit})\ \text{ann-lits}$  **and**  
 $\text{clauses}_{NOT} :: 'st \Rightarrow 'v\ \text{clauses}$  **and**

```

    prepend-trail :: ('v, unit) ann-lit ⇒ 'st ⇒ 'st and
    tl-trail :: 'st ⇒ 'st and
    add-clsNOT :: 'v clause ⇒ 'st ⇒ 'st and
    remove-clsNOT :: 'v clause ⇒ 'st ⇒ 'st
begin
abbreviation stateNOT :: 'st ⇒ ('v, unit) ann-lit list × 'v clauses where
stateNOT S ≡ (trail S, clausesNOT 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 clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT — related to the state
for
  trail :: 'st ⇒ ('v, unit) ann-lits and
  clausesNOT :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit) ann-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT :: 'v clause ⇒ 'st ⇒ 'st and
  remove-clsNOT :: 'v clause ⇒ 'st ⇒ 'st +
assumes
  prepend-trailNOT:
    stateNOT (prepend-trail L st) = (L # trail st, clausesNOT st) and
  tl-trailNOT:
    stateNOT (tl-trail st) = (tl (trail st), clausesNOT st) and
  add-clsNOT:
    stateNOT (add-clsNOT C st) = (trail st, {#C#} + clausesNOT st) and
  remove-clsNOT:
    stateNOT (remove-clsNOT C st) = (trail st, removeAll-mset C (clausesNOT st))
begin
lemma
  trail-prepend-trail[simp]:
    trail (prepend-trail L st) = L # trail st
  and
  trail-tl-trailNOT[simp]: trail (tl-trail st) = tl (trail st) and
  trail-add-clsNOT[simp]: trail (add-clsNOT C st) = trail st and
  trail-remove-clsNOT[simp]: trail (remove-clsNOT C st) = trail st and

  clauses-prepend-trail[simp]:
    clausesNOT (prepend-trail L st) = clausesNOT st
  and
  clauses-tl-trail[simp]: clausesNOT (tl-trail st) = clausesNOT st and
  clauses-add-clsNOT[simp]:
    clausesNOT (add-clsNOT C st) = {#C#} + clausesNOT st and
  clauses-remove-clsNOT[simp]:
    clausesNOT (remove-clsNOT C st) = removeAll-mset C (clausesNOT st)
  ⟨proof⟩

```

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

```

function reduce-trail-toNOT :: 'a list ⇒ 'st ⇒ 'st where
reduce-trail-toNOT F S =
  (if length (trail S) = length F ∨ trail S = [] then S else reduce-trail-toNOT F (tl-trail S))
  ⟨proof⟩
termination ⟨proof⟩

```



**declare** *reduce-trail-to<sub>NOT</sub>.simps*[simp del]

Then we need several lemmas about the *reduce-trail-to<sub>NOT</sub>*.

**lemma**

**shows**

*reduce-trail-to<sub>NOT</sub>-Nil*[simp]:  $\text{trail } S = [] \implies \text{reduce-trail-to}_{NOT} F S = S$  **and**  
*reduce-trail-to<sub>NOT</sub>-eq-length*[simp]:  $\text{length } (\text{trail } S) = \text{length } F \implies \text{reduce-trail-to}_{NOT} F S = S$   
 ⟨proof⟩

**lemma** *reduce-trail-to<sub>NOT</sub>-length-ne*[simp]:

$\text{length } (\text{trail } S) \neq \text{length } F \implies \text{trail } S \neq [] \implies$   
 $\text{reduce-trail-to}_{NOT} F S = \text{reduce-trail-to}_{NOT} F (\text{tl-trail } S)$   
 ⟨proof⟩

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-length-le*:

**assumes**  $\text{length } F > \text{length } (\text{trail } S)$   
**shows**  $\text{trail } (\text{reduce-trail-to}_{NOT} F S) = []$   
 ⟨proof⟩

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-Nil*[simp]:

$\text{trail } (\text{reduce-trail-to}_{NOT} [] S) = []$   
 ⟨proof⟩

**lemma** *clauses-reduce-trail-to<sub>NOT</sub>-Nil*:

$\text{clauses}_{NOT} (\text{reduce-trail-to}_{NOT} [] S) = \text{clauses}_{NOT} S$   
 ⟨proof⟩

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-drop*:

$\text{trail } (\text{reduce-trail-to}_{NOT} F S) =$   
 (if  $\text{length } (\text{trail } S) \geq \text{length } F$   
 then  $\text{drop } (\text{length } (\text{trail } S) - \text{length } F) (\text{trail } S)$   
 else  $[]$ )  
 ⟨proof⟩

**lemma** *reduce-trail-to<sub>NOT</sub>-skip-beginning*:

**assumes**  $\text{trail } S = F' @ F$   
**shows**  $\text{trail } (\text{reduce-trail-to}_{NOT} F S) = F$   
 ⟨proof⟩

**lemma** *reduce-trail-to<sub>NOT</sub>-clauses*[simp]:

$\text{clauses}_{NOT} (\text{reduce-trail-to}_{NOT} F S) = \text{clauses}_{NOT} S$   
 ⟨proof⟩

**lemma** *trail-eq-reduce-trail-to<sub>NOT</sub>-eq*:

$\text{trail } S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to}_{NOT} F S) = \text{trail } (\text{reduce-trail-to}_{NOT} F T)$   
 ⟨proof⟩

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-add-cl<sub>NOT</sub>*[simp]:

$\text{no-dup } (\text{trail } S) \implies$   
 $\text{trail } (\text{reduce-trail-to}_{NOT} F (\text{add-cl}_{NOT} C S)) = \text{trail } (\text{reduce-trail-to}_{NOT} F S)$   
 ⟨proof⟩

**lemma** *reduce-trail-to<sub>NOT</sub>-trail-tl-trail-decomp*[simp]:

$\text{trail } S = F' @ \text{Decided } K \# F \implies$   
 $\text{trail } (\text{reduce-trail-to}_{NOT} F (\text{tl-trail } S)) = F$   
 ⟨proof⟩

**lemma** *reduce-trail-to<sub>NOT</sub>-length*:

$length\ M = length\ M' \implies reduce-trail-to_{NOT}\ M\ S = reduce-trail-to_{NOT}\ M'\ S$   
 $\langle proof \rangle$

**abbreviation** *trail-weight* **where**

$trail-weight\ S \equiv map\ ((\lambda l.\ 1 + length\ l)\ o\ snd)\ (get-all-ann-decomposition\ (trail\ S))$

As we are defining abstract states, the Isabelle equality about them is too strong: we want the weaker equivalence stating that two states are equal if they cannot be distinguished, i.e. given the getter *trail* and *clauses<sub>NOT</sub>* do not distinguish them.

**definition** *state-eq<sub>NOT</sub>* ::  $'st \Rightarrow 'st \Rightarrow bool$  (**infix**  $\sim 50$ ) **where**

$S \sim T \longleftrightarrow trail\ S = trail\ T \wedge clauses_{NOT}\ S = clauses_{NOT}\ T$

**lemma** *state-eq<sub>NOT</sub>-ref[simp]*:

$S \sim S$

$\langle proof \rangle$

**lemma** *state-eq<sub>NOT</sub>-sym*:

$S \sim T \longleftrightarrow T \sim S$

$\langle proof \rangle$

**lemma** *state-eq<sub>NOT</sub>-trans*:

$S \sim T \implies T \sim U \implies S \sim U$

$\langle proof \rangle$

**lemma**

**shows**

*state-eq<sub>NOT</sub>-trail*:  $S \sim T \implies trail\ S = trail\ T$  **and**

*state-eq<sub>NOT</sub>-clauses*:  $S \sim T \implies clauses_{NOT}\ S = clauses_{NOT}\ T$

$\langle proof \rangle$

**lemmas** *state-simp<sub>NOT</sub>[simp]* = *state-eq<sub>NOT</sub>-trail* *state-eq<sub>NOT</sub>-clauses*

**lemma** *reduce-trail-to<sub>NOT</sub>-state-eq<sub>NOT</sub>-compatible*:

**assumes** *ST*:  $S \sim T$

**shows** *reduce-trail-to<sub>NOT</sub>*  $F\ S \sim reduce-trail-to_{NOT}\ F\ T$

$\langle proof \rangle$

**end**

## Definition of the operation

Each possible is in its own locale.

**locale** *propagate-ops* =

*dpll-state* *trail* *clauses<sub>NOT</sub>* *prepend-trail* *tl-trail* *add-cl<sub>s</sub><sub>NOT</sub>* *remove-cl<sub>s</sub><sub>NOT</sub>*

**for**

*trail* ::  $'st \Rightarrow ('v, unit)\ ann-lits$  **and**

*clauses<sub>NOT</sub>* ::  $'st \Rightarrow 'v\ clauses$  **and**

*prepend-trail* ::  $('v, unit)\ ann-lit \Rightarrow 'st \Rightarrow 'st$  **and**

*tl-trail* ::  $'st \Rightarrow 'st$  **and**

*add-cl<sub>s</sub><sub>NOT</sub>* ::  $'v\ clause \Rightarrow 'st \Rightarrow 'st$  **and**

*remove-cl<sub>s</sub><sub>NOT</sub>* ::  $'v\ clause \Rightarrow 'st \Rightarrow 'st +$

**fixes**

*propagate-cond* ::  $('v, unit)\ ann-lit \Rightarrow 'st \Rightarrow bool$

```

begin
inductive propagateNOT :: 'st ⇒ 'st ⇒ bool where
propagateNOT[intro]:  $C + \{\#L\} \in \# \text{ clauses}_{NOT} S \implies \text{trail } S \models_{as} CNot C$ 
    ⇒ undefined-lit (trail S) L
    ⇒ propagate-cond (Propagated L ()) S
    ⇒  $T \sim \text{prepend-trail} (\text{Propagated } L ()) S$ 
    ⇒ propagateNOT S T
inductive-cases propagateNOTE[elim]: propagateNOT S T

end

locale decide-ops =
  dpll-state trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
for
  trail :: 'st ⇒ ('v, unit) ann-lits and
  clausesNOT :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit) ann-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clNOT :: 'v clause ⇒ 'st ⇒ 'st and
  remove-clNOT :: 'v clause ⇒ 'st ⇒ 'st
begin
inductive decideNOT :: 'st ⇒ 'st ⇒ bool where
decideNOT[intro]: undefined-lit (trail S) L ⇒ atm-of L ∈ atms-of-mm (clausesNOT S)
    ⇒  $T \sim \text{prepend-trail} (\text{Decided } L) S$ 
    ⇒ decideNOT S T
inductive-cases decideNOTE[elim]: decideNOT S S'
end

locale backjumping-ops =
  dpll-state trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
for
  trail :: 'st ⇒ ('v, unit) ann-lits and
  clausesNOT :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit) ann-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clNOT :: 'v clause ⇒ 'st ⇒ 'st and
  remove-clNOT :: 'v clause ⇒ 'st ⇒ 'st +
fixes
  backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool
begin

inductive backjump where
trail S = F' @ Decided K# F
    ⇒  $T \sim \text{prepend-trail} (\text{Propagated } L ()) (\text{reduce-trail-to}_{NOT} F S)$ 
    ⇒  $C \in \# \text{ clauses}_{NOT} S$ 
    ⇒  $\text{trail } S \models_{as} CNot C$ 
    ⇒ undefined-lit F L
    ⇒  $\text{atm-of } L \in \text{atms-of-mm} (\text{clauses}_{NOT} S) \cup \text{atm-of ' (lits-of-l (trail } S))$ 
    ⇒  $\text{clauses}_{NOT} S \models_{pm} C' + \{\#L\}$ 
    ⇒  $F \models_{as} CNot C'$ 
    ⇒ backjump-conds C C' L S T
    ⇒ backjump S T
inductive-cases backjumpE: backjump S T

```

The condition  $\text{atm-of } L \in \text{atms-of-mm} (\text{clauses}_{NOT} S) \cup \text{atm-of ' (lits-of-l (trail } S))$  is not

implied by 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 clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds +
  decide-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT +
  backjumping-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT backjump-conds
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool +
assumes
  bj-can-jump:
   $\bigwedge S C F' K F L.$ 
  inv  $S \Rightarrow$ 
  no-dup (trail  $S$ )  $\Rightarrow$ 
  trail  $S = F' @ Decided K \# F \Rightarrow$ 
   $C \in \# clauses_{NOT} S \Rightarrow$ 
  trail  $S \models_{as} CNot C \Rightarrow$ 
  undefined-lit  $F L \Rightarrow$ 
  atm-of  $L \in atms-of-mm (clauses_{NOT} S) \cup atm-of (lits-of-l (F' @ Decided K \# F)) \Rightarrow$ 
   $clauses_{NOT} S \models_{pm} C' + \{\#L\# \} \Rightarrow$ 
   $F \models_{as} CNot C' \Rightarrow$ 
   $\neg no-step backjump S$ 
begin

```

We cannot add a like condition  $atms-of C' \subseteq atms-of-ms N$  to ensure that we can backjump even if the last decision variable has disappeared from the set of clauses.

The part of the condition  $atm-of L \in atm-of (lits-of-l (F' @ Decided K \# F))$  is important, otherwise you are not sure that you can backtrack.

#### Definition

We define dpll with backjumping:

```

inductive dpll-bj :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for  $S :: 'st$  where
  bj-decideNOT: decideNOT  $S S' \Rightarrow dpll-bj S S' |$ 
  bj-propagateNOT: propagateNOT  $S S' \Rightarrow dpll-bj S S' |$ 
  bj-backjump: backjump  $S S' \Rightarrow dpll-bj S S'$ 

```

**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*<sub>NOT</sub> *propagate*<sub>NOT</sub> *backjump*]:

```

fixes  $S T :: 'st$ 
assumes
  dpll-bj  $S T$  and
  inv  $S$ 

```

$\wedge L \ T. \text{ undefined-lit } (\text{trail } S) \ L \implies \text{atm-of } L \in \text{atms-of-mm } (\text{clauses}_{NOT} \ S)$   
 $\implies T \sim \text{prepend-trail } (\text{Decided } L) \ S$   
 $\implies P \ S \ T \text{ and}$   
 $\wedge C \ L \ T. \ C + \{\#L\# \} \in \# \text{ clauses}_{NOT} \ S \implies \text{trail } S \models_{as} CNot \ C \implies \text{undefined-lit } (\text{trail } S) \ L$   
 $\implies T \sim \text{prepend-trail } (\text{Propagated } L \ ()) \ S$   
 $\implies P \ S \ T \text{ and}$   
 $\wedge C \ F' \ K \ F \ L \ C' \ T. \ C \in \# \text{ clauses}_{NOT} \ S \implies F' @ \text{Decided } K \ \# \ F \models_{as} CNot \ C$   
 $\implies \text{trail } S = F' @ \text{Decided } K \ \# \ F$   
 $\implies \text{undefined-lit } F \ L$   
 $\implies \text{atm-of } L \in \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \cup \text{atm-of } ' (\text{lits-of-l } (F' @ \text{Decided } K \ \# \ F))$   
 $\implies \text{clauses}_{NOT} \ S \models_{pm} C' + \{\#L\# \}$   
 $\implies F \models_{as} CNot \ C'$   
 $\implies T \sim \text{prepend-trail } (\text{Propagated } L \ ()) \ (\text{reduce-trail-to}_{NOT} \ F \ S)$   
 $\implies P \ S \ T$   
**shows**  $P \ S \ T$   
 $\langle \text{proof} \rangle$

## Basic properties

**First, some better suited induction principle** lemma *dpll-bj-clauses*:

**assumes**  $dpll\text{-}bj \ S \ T$  **and**  $inv \ S$   
**shows**  $\text{clauses}_{NOT} \ S = \text{clauses}_{NOT} \ T$   
 $\langle \text{proof} \rangle$

**No duplicates in the trail** lemma *dpll-bj-no-dup*:

**assumes**  $dpll\text{-}bj \ S \ T$  **and**  $inv \ S$   
**and**  $no\text{-}dup \ (\text{trail } S)$   
**shows**  $no\text{-}dup \ (\text{trail } T)$   
 $\langle \text{proof} \rangle$

**Valuations** lemma *dpll-bj-sat-iff*:

**assumes**  $dpll\text{-}bj \ S \ T$  **and**  $inv \ S$   
**shows**  $I \models_{sm} \text{clauses}_{NOT} \ S \longleftrightarrow I \models_{sm} \text{clauses}_{NOT} \ T$   
 $\langle \text{proof} \rangle$

**Clauses** lemma *dpll-bj-atms-of-ms-clauses-inv*:

**assumes**  
 $dpll\text{-}bj \ S \ T$  **and**  
 $inv \ S$   
**shows**  $\text{atms-of-mm } (\text{clauses}_{NOT} \ S) = \text{atms-of-mm } (\text{clauses}_{NOT} \ T)$   
 $\langle \text{proof} \rangle$

**lemma** *dpll-bj-atms-in-trail*:

**assumes**  
 $dpll\text{-}bj \ S \ T$  **and**  
 $inv \ S$  **and**  
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} \ S)$   
**shows**  $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} \ S)$   
 $\langle \text{proof} \rangle$

**lemma** *dpll-bj-atms-in-trail-in-set*:

**assumes**  $dpll\text{-}bj \ S \ T$  **and**  
 $inv \ S$  **and**  
 $\text{atms-of-mm } (\text{clauses}_{NOT} \ S) \subseteq A$  **and**  
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq A$

**shows**  $\text{atm-of } \text{' (lits-of-l (trail } T)) \subseteq A$   
 $\langle \text{proof} \rangle$

**lemma** *dpll-bj-all-decomposition-implies-inv*:

**assumes**  
 $\text{dpll-bj } S \ T$  **and**  
 $\text{inv: inv } S$  **and**  
 $\text{decomp: all-decomposition-implies-m (clauses}_{\text{NOT}} S) (\text{get-all-ann-decomposition (trail } S))$   
**shows**  $\text{all-decomposition-implies-m (clauses}_{\text{NOT}} T) (\text{get-all-ann-decomposition (trail } T))$   
 $\langle \text{proof} \rangle$

## Termination

**Using a proper measure lemma** *length-get-all-ann-decomposition-append-Decided*:

$\text{length (get-all-ann-decomposition (F' @ Decided K \# F))} =$   
 $\text{length (get-all-ann-decomposition F')}$   
 $+ \text{length (get-all-ann-decomposition (Decided K \# F))}$   
 $- 1$   
 $\langle \text{proof} \rangle$

**lemma** *take-length-get-all-ann-decomposition-decided-sandwich*:

$\text{take (length (get-all-ann-decomposition F))}$   
 $(\text{map (f o snd) (rev (get-all-ann-decomposition (F' @ Decided K \# F))))$   
 $=$   
 $\text{map (f o snd) (rev (get-all-ann-decomposition F))}$

$\langle \text{proof} \rangle$

**lemma** *length-get-all-ann-decomposition-length*:

$\text{length (get-all-ann-decomposition } M) \leq 1 + \text{length } M$   
 $\langle \text{proof} \rangle$

**lemma** *length-in-get-all-ann-decomposition-bounded*:

**assumes**  $i: i \in \text{set (trail-weight } S)$   
**shows**  $i \leq \text{Suc (length (trail } S))$   
 $\langle \text{proof} \rangle$

**Well-foundedness** The bounds are the following:

- $1 + \text{card (atms-of-ms } A)$ :  $\text{card (atms-of-ms } A)$  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.
- $2 + \text{card (atms-of-ms } A)$ :  $\text{card (atms-of-ms } A)$  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* ::  $'b \text{ literal multiset set} \Rightarrow 'a \text{ list} \Rightarrow \text{nat}$  **where**

$\text{unassigned-lit } N \ M \equiv \text{card (atms-of-ms } N) - \text{length } M$

**lemma** *dpll-bj-trail-mes-increasing-prop*:

**fixes**  $M :: ('v, \text{unit}) \text{ ann-lits}$  **and**  $N :: 'v \text{ clauses}$   
**assumes**  
 $\text{dpll-bj } S \ T$  **and**  
 $\text{inv } S$  **and**  
 $\text{NA: atms-of-mm (clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A$  **and**

$MA: atm\text{-}of \text{ ' } lits\text{-}of\text{-}l (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $n\text{-}d: no\text{-}dup (trail\ S)$  **and**  
 $finite: finite\ A$   
**shows**  $\mu_C (1 + card (atms\text{-}of\text{-}ms\ A)) (2 + card (atms\text{-}of\text{-}ms\ A)) (trail\text{-}weight\ T)$   
 $> \mu_C (1 + card (atms\text{-}of\text{-}ms\ A)) (2 + card (atms\text{-}of\text{-}ms\ A)) (trail\text{-}weight\ S)$   
 $\langle proof \rangle$

**lemma** *dpll-bj-trail-mes-decreasing-prop*:  
**assumes**  $dpll: dpll\text{-}bj\ S\ T$  **and**  $inv: inv\ S$  **and**  
 $N\text{-}A: atms\text{-}of\text{-}mm (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $M\text{-}A: atm\text{-}of \text{ ' } lits\text{-}of\text{-}l (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $nd: no\text{-}dup (trail\ S)$  **and**  
 $fin\text{-}A: finite\ A$   
**shows**  $(2 + card (atms\text{-}of\text{-}ms\ A)) \wedge (1 + card (atms\text{-}of\text{-}ms\ A))$   
 $\quad - \mu_C (1 + card (atms\text{-}of\text{-}ms\ A)) (2 + card (atms\text{-}of\text{-}ms\ A)) (trail\text{-}weight\ T)$   
 $< (2 + card (atms\text{-}of\text{-}ms\ A)) \wedge (1 + card (atms\text{-}of\text{-}ms\ A))$   
 $\quad - \mu_C (1 + card (atms\text{-}of\text{-}ms\ A)) (2 + card (atms\text{-}of\text{-}ms\ A)) (trail\text{-}weight\ S)$   
 $\langle proof \rangle$

**lemma** *wf-dpll-bj*:  
**assumes**  $fin: finite\ A$   
**shows**  $wf \{(T, S). dpll\text{-}bj\ S\ T$   
 $\quad \wedge atms\text{-}of\text{-}mm (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of \text{ ' } lits\text{-}of\text{-}l (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$   
 $\quad \wedge no\text{-}dup (trail\ S) \wedge inv\ S\}$   
 $(is\ wf\ ?A)$   
 $\langle proof \rangle$

## 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 that *satisfiable*  $N$ ,  $\neg M \models_{as} N$  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  $\neg M \models_{as} N$  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*  $dpll\text{-}bj\ S$

**theorem** *dpll-backjump-final-state*:  
**fixes**  $A :: 'v\ clause\ set$  **and**  $S\ T :: 'st$   
**assumes**  
 $atms\text{-}of\text{-}mm (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $atm\text{-}of \text{ ' } lits\text{-}of\text{-}l (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $no\text{-}dup (trail\ S)$  **and**  
 $finite\ A$  **and**  
 $inv: inv\ S$  **and**

*n-s: no-step dpll-bj S and*  
*decomp: all-decomposition-implies-m (clauses<sub>NOT</sub> S) (get-all-ann-decomposition (trail S))*  
**shows** *unsatisfiable (set-mset (clauses<sub>NOT</sub> S))*  
 $\vee$  *(trail S  $\models_{asm}$  clauses<sub>NOT</sub> S  $\wedge$  satisfiable (set-mset (clauses<sub>NOT</sub> S)))*  
 $\langle proof \rangle$

**end** — End of *dpll-with-backjumping-ops*

**locale** *dpll-with-backjumping* =  
*dpll-with-backjumping-ops trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> inv*  
*backjump-conds propagate-conds*  
**for**  
*trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and*  
*clauses<sub>NOT</sub> :: 'st  $\Rightarrow$  'v clauses and*  
*prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*tl-trail :: 'st  $\Rightarrow$  'st and*  
*add-cl<sub>NOT</sub> :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*remove-cl<sub>NOT</sub> :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*inv :: 'st  $\Rightarrow$  bool and*  
*backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and*  
*propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool*  
 $+$   
**assumes** *dpll-bj-inv:  $\bigwedge S T. dpll-bj S T \Rightarrow inv S \Rightarrow inv T$*   
**begin**

**lemma** *rtrancpl-dpll-bj-inv:*  
**assumes** *dpll-bj\*\* S T and inv S*  
**shows** *inv T*  
 $\langle proof \rangle$

**lemma** *rtrancpl-dpll-bj-no-dup:*  
**assumes** *dpll-bj\*\* S T and inv S*  
**and** *no-dup (trail S)*  
**shows** *no-dup (trail T)*  
 $\langle proof \rangle$

**lemma** *rtrancpl-dpll-bj-atms-of-ms-clauses-inv:*  
**assumes**  
*dpll-bj\*\* S T and inv S*  
**shows** *atms-of-mm (clauses<sub>NOT</sub> S) = atms-of-mm (clauses<sub>NOT</sub> T)*  
 $\langle proof \rangle$

**lemma** *rtrancpl-dpll-bj-atms-in-trail:*  
**assumes**  
*dpll-bj\*\* S T and*  
*inv S and*  
*atm-of ' (lits-of-l (trail S))  $\subseteq$  atms-of-mm (clauses<sub>NOT</sub> S)*  
**shows** *atm-of ' (lits-of-l (trail T))  $\subseteq$  atms-of-mm (clauses<sub>NOT</sub> T)*  
 $\langle proof \rangle$

**lemma** *rtrancpl-dpll-bj-sat-iff:*  
**assumes** *dpll-bj\*\* S T and inv S*  
**shows**  *$I \models_{sm} clauses_{NOT} S \longleftrightarrow I \models_{sm} clauses_{NOT} T$*   
 $\langle proof \rangle$

**lemma** *rtrancpl-dpll-bj-atms-in-trail-in-set:*



**assumes**  
*dpll-bj\*\* S T and*  
*inv S*  
*atms-of-mm (clauses<sub>NOT</sub> S) ⊆ A and*  
*atm-of ‘ (lits-of-l (trail S)) ⊆ A*  
**shows** *atm-of ‘ (lits-of-l (trail T)) ⊆ A*  
 ⟨proof⟩

**lemma** *rtrancpl-dpll-bj-all-decomposition-implies-inv:*

**assumes**  
*dpll-bj\*\* S T and*  
*inv S*  
*all-decomposition-implies-m (clauses<sub>NOT</sub> S) (get-all-ann-decomposition (trail S))*  
**shows** *all-decomposition-implies-m (clauses<sub>NOT</sub> T) (get-all-ann-decomposition (trail T))*  
 ⟨proof⟩

**lemma** *rtrancpl-dpll-bj-inv-incl-dpll-bj-inv-tranc:*

{(T, S). *dpll-bj<sup>++</sup> S T*  
 ∧ *atms-of-mm (clauses<sub>NOT</sub> 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<sub>NOT</sub> S) ⊆ atms-of-ms A*  
 ∧ *atm-of ‘ lits-of-l (trail S) ⊆ atms-of-ms A* ∧ *no-dup (trail S) ∧ inv S*}<sup>+</sup>  
 (is ?A ⊆ ?B<sup>+</sup>)  
 ⟨proof⟩

**lemma** *wf-trancpl-dpll-bj:*

**assumes** *fin: finite A*  
**shows** *wf {(T, S). dpll-bj<sup>++</sup> S T*  
 ∧ *atms-of-mm (clauses<sub>NOT</sub> 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 ⊨<sub>sextm</sub> clauses<sub>NOT</sub> S ⇔ I ⊨<sub>sextm</sub> clauses<sub>NOT</sub> T*  
 ⟨proof⟩

**lemma** *rtrancpl-dpll-bj-sat-ext-iff:*

*dpll-bj\*\* S T ⇒ inv S ⇒ I ⊨<sub>sextm</sub> clauses<sub>NOT</sub> S ⇔ I ⊨<sub>sextm</sub> clauses<sub>NOT</sub> 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<sub>NOT</sub> 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<sub>NOT</sub> S) (get-all-ann-decomposition (trail S))*  
**shows** *unsatisfiable (set-mset (clauses<sub>NOT</sub> S))*  
 ∨ *(trail T ⊨<sub>asm</sub> clauses<sub>NOT</sub> S ∧ satisfiable (set-mset (clauses<sub>NOT</sub> 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*<sub>NOT</sub> *S* = *N* **and**  
*inv* *S*  
**shows** *unsatisfiable* (*set-mset* *N*)  $\vee$  (*trail* *T*  $\models_{asm}$  *N*  $\wedge$  *satisfiable* (*set-mset* *N*))  
 $\langle proof \rangle$

**lemma** *trancpl-dpll-bj-trail-mes-decreasing-prop*:

**assumes** *dpll*: *dpll-bj*<sup>++</sup> *S T* **and** *inv*: *inv* *S* **and**  
*N-A*: *atms-of-mm* (*clauses*<sub>NOT</sub> *S*)  $\subseteq$  *atms-of-ms* *A* **and**  
*M-A*: *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**  $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$   
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$   
 $\quad < (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$   
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$   
 $\langle proof \rangle$

**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*<sub>NOT</sub> *prepend-trail* *tl-trail* *add-cls*<sub>NOT</sub> *remove-cls*<sub>NOT</sub>  
**for**  
*trail* :: '*st*  $\Rightarrow$  ('*v*, *unit*) *ann-lits* **and**  
*clauses*<sub>NOT</sub> :: '*st*  $\Rightarrow$  '*v* *clauses* **and**  
*prepend-trail* :: ('*v*, *unit*) *ann-lit*  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* **and**  
*tl-trail* :: '*st*  $\Rightarrow$  '*st* **and**  
*add-cls*<sub>NOT</sub> :: '*v* *clause*  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* **and**  
*remove-cls*<sub>NOT</sub> :: '*v* *clause*  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* +  
**fixes**  
*learn-cond* :: '*v* *clause*  $\Rightarrow$  '*st*  $\Rightarrow$  *bool*  
**begin**  
**inductive** *learn* :: '*st*  $\Rightarrow$  '*st*  $\Rightarrow$  *bool* **where**  
*learn*<sub>NOT</sub>-rule: *clauses*<sub>NOT</sub> *S*  $\models_{pm}$  *C*  $\Rightarrow$   
 $\text{atms-of } C \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of ' } (\text{lits-of-l } (\text{trail } S)) \Rightarrow$   
 $\text{learn-cond } C S \Rightarrow$   
 $T \sim \text{add-cl}_{NOT} C S \Rightarrow$   
 $\text{learn } S T$   
**inductive-cases** *learn*<sub>NOT</sub>*E*: *learn* *S T*

**lemma** *learn- $\mu_C$ -stable*:

**assumes** *learn* *S T* **and** *no-dup* (*trail* *S*)  
**shows**  $\mu_C A B (\text{trail-weight } S) = \mu_C A B (\text{trail-weight } T)$   
 $\langle proof \rangle$

**end**

Forget removes an information that can be deduced from the context (e.g. redundant clauses, tautologies)

**locale** *forget-ops* =  
*dpll-state* *trail* *clauses*<sub>NOT</sub> *prepend-trail* *tl-trail* *add-cls*<sub>NOT</sub> *remove-cls*<sub>NOT</sub>  
**for**  
*trail* :: 'st  $\Rightarrow$  ('v, unit) *ann-lits* **and**  
*clauses*<sub>NOT</sub> :: 'st  $\Rightarrow$  'v *clauses* **and**  
*prepend-trail* :: ('v, unit) *ann-lit*  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-cls*<sub>NOT</sub> :: 'v *clause*  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-cls*<sub>NOT</sub> :: 'v *clause*  $\Rightarrow$  'st  $\Rightarrow$  'st +  
**fixes**  
*forget-cond* :: 'v *clause*  $\Rightarrow$  'st  $\Rightarrow$  bool  
**begin**  
**inductive** *forget*<sub>NOT</sub> :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **where**  
*forget*<sub>NOT</sub>:  
*removeAll-mset* *C* (*clauses*<sub>NOT</sub> *S*)  $\models_{pm}$  *C*  $\Rightarrow$   
*forget-cond* *C* *S*  $\Rightarrow$   
*C*  $\in \#$  *clauses*<sub>NOT</sub> *S*  $\Rightarrow$   
*T*  $\sim$  *remove-cls*<sub>NOT</sub> *C* *S*  $\Rightarrow$   
*forget*<sub>NOT</sub> *S* *T*  
**inductive-cases** *forget*<sub>NOT</sub>*E*: *forget*<sub>NOT</sub> *S* *T*

**lemma** *forget- $\mu_C$ -stable*:  
**assumes** *forget*<sub>NOT</sub> *S* *T*  
**shows**  $\mu_C$  *A* *B* (*trail-weight* *S*) =  $\mu_C$  *A* *B* (*trail-weight* *T*)  
*<proof>*  
**end**

**locale** *learn-and-forget*<sub>NOT</sub> =  
*learn-ops* *trail* *clauses*<sub>NOT</sub> *prepend-trail* *tl-trail* *add-cls*<sub>NOT</sub> *remove-cls*<sub>NOT</sub> *learn-cond* +  
*forget-ops* *trail* *clauses*<sub>NOT</sub> *prepend-trail* *tl-trail* *add-cls*<sub>NOT</sub> *remove-cls*<sub>NOT</sub> *forget-cond*  
**for**  
*trail* :: 'st  $\Rightarrow$  ('v, unit) *ann-lits* **and**  
*clauses*<sub>NOT</sub> :: 'st  $\Rightarrow$  'v *clauses* **and**  
*prepend-trail* :: ('v, unit) *ann-lit*  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-cls*<sub>NOT</sub> :: 'v *clause*  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-cls*<sub>NOT</sub> :: 'v *clause*  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*learn-cond* *forget-cond* :: 'v *clause*  $\Rightarrow$  'st  $\Rightarrow$  bool  
**begin**  
**inductive** *learn-and-forget*<sub>NOT</sub> :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool  
**where**  
*lf-learn*: *learn* *S* *T*  $\Rightarrow$  *learn-and-forget*<sub>NOT</sub> *S* *T* |  
*lf-forget*: *forget*<sub>NOT</sub> *S* *T*  $\Rightarrow$  *learn-and-forget*<sub>NOT</sub> *S* *T*  
**end**

## Definition of CDCL

**locale** *conflict-driven-clause-learning-ops* =  
*dpll-with-backjumping-ops* *trail* *clauses*<sub>NOT</sub> *prepend-trail* *tl-trail* *add-cls*<sub>NOT</sub> *remove-cls*<sub>NOT</sub>  
*inv* *backjump-conds* *propagate-conds* +  
*learn-and-forget*<sub>NOT</sub> *trail* *clauses*<sub>NOT</sub> *prepend-trail* *tl-trail* *add-cls*<sub>NOT</sub> *remove-cls*<sub>NOT</sub> *learn-cond*  
*forget-cond*

**for**

$trail :: 'st \Rightarrow ('v, unit) \text{ ann-lits and}$   
 $clauses_{NOT} :: 'st \Rightarrow 'v \text{ clauses and}$   
 $prepend-trail :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $tl-trail :: 'st \Rightarrow 'st \text{ and}$   
 $add-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $remove-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $inv :: 'st \Rightarrow bool \text{ and}$   
 $backjump-conds :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow bool \text{ and}$   
 $propagate-conds :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow bool \text{ and}$   
 $learn-cond \text{ forget-cond} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow bool$

**begin**

**inductive**  $cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$  **for**  $S :: 'st$  **where**

$c\text{-dpll-bj}: dpll\text{-bj } S S' \Longrightarrow cdcl_{NOT} S S' \mid$   
 $c\text{-learn}: learn S S' \Longrightarrow cdcl_{NOT} S S' \mid$   
 $c\text{-forget}_{NOT}: forget_{NOT} S S' \Longrightarrow cdcl_{NOT} S S'$

**lemma**  $cdcl_{NOT}\text{-all-induct}[consumes 1, \text{case-names } dpll\text{-bj } learn \text{ forget}_{NOT}]$ :

**fixes**  $S T :: 'st$

**assumes**  $cdcl_{NOT} S T$  **and**

$dpll: \bigwedge T. dpll\text{-bj } S T \Longrightarrow P S T$  **and**

*learning:*

$\bigwedge C T. clauses_{NOT} S \models_{pm} C \Longrightarrow$   
 $atms\text{-of } C \subseteq atms\text{-of-mm } (clauses_{NOT} S) \cup atm\text{-of } (lits\text{-of-l } (trail S)) \Longrightarrow$   
 $T \sim add\text{-cls}_{NOT} C S \Longrightarrow$   
 $P S T$  **and**

*forgetting:*  $\bigwedge C T. removeAll\text{-mset } C (clauses_{NOT} S) \models_{pm} C \Longrightarrow$

$C \in \# clauses_{NOT} S \Longrightarrow$   
 $T \sim remove\text{-cls}_{NOT} C S \Longrightarrow$   
 $P S T$

**shows**  $P S T$

$\langle proof \rangle$

**lemma**  $cdcl_{NOT}\text{-no-dup}$ :

**assumes**

$cdcl_{NOT} S T$  **and**

$inv S$  **and**

$no\text{-dup } (trail S)$

**shows**  $no\text{-dup } (trail T)$

$\langle proof \rangle$

**Consistency of the trail lemma**  $cdcl_{NOT}\text{-consistent}$ :

**assumes**

$cdcl_{NOT} S T$  **and**

$inv S$  **and**

$no\text{-dup } (trail S)$

**shows**  $consistent\text{-interp } (lits\text{-of-l } (trail T))$

$\langle proof \rangle$

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**  $cdcl_{NOT}\text{-atms-of-ms-clauses-decreasing}$ :

**assumes**  $cdcl_{NOT} S T$  **and**  $inv S$  **and**  $no\text{-dup } (trail S)$

**shows**  $\text{atms-of-mm } (\text{clauses}_{NOT} T) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_{NOT}\text{-atms-in-trail}$ :

**assumes**  $\text{cdcl}_{NOT} S T$  **and**  $\text{inv } S$  **and**  $\text{no-dup } (\text{trail } S)$   
**and**  $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S)$   
**shows**  $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_{NOT}\text{-atms-in-trail-in-set}$ :

**assumes**  
 $\text{cdcl}_{NOT} S T$  **and**  $\text{inv } S$  **and**  $\text{no-dup } (\text{trail } S)$  **and**  
 $\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq A$  **and**  
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq A$   
**shows**  $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq A$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_{NOT}\text{-all-decomposition-implies}$ :

**assumes**  $\text{cdcl}_{NOT} S T$  **and**  $\text{inv } S$  **and**  $n\text{-d}[\text{simp}]: \text{no-dup } (\text{trail } S)$  **and**  
 $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} S) (\text{get-all-ann-decomposition } (\text{trail } S))$   
**shows**  
 $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} T) (\text{get-all-ann-decomposition } (\text{trail } T))$   
 $\langle \text{proof} \rangle$

**Extension of models** **lemma**  $\text{cdcl}_{NOT}\text{-bj-sat-ext-iff}$ :

**assumes**  $\text{cdcl}_{NOT} S T$  **and**  $\text{inv } S$  **and**  $n\text{-d}: \text{no-dup } (\text{trail } S)$   
**shows**  $I \models_{\text{sextm}} \text{clauses}_{NOT} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} T$   
 $\langle \text{proof} \rangle$

**end** — end of *conflict-driven-clause-learning-ops*

## CDCL with invariant

**locale**  $\text{conflict-driven-clause-learning} =$   
 $\text{conflict-driven-clause-learning-ops} +$   
**assumes**  $\text{cdcl}_{NOT}\text{-inv}: \bigwedge S T. \text{cdcl}_{NOT} S T \implies \text{inv } S \implies \text{inv } T$   
**begin**  
**sublocale**  $\text{dpll-with-backjumping}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-inv}$ :

$\text{cdcl}_{NOT}^{**} S T \implies \text{inv } S \implies \text{inv } T$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-no-dup}$ :

**assumes**  $\text{cdcl}_{NOT}^{**} S T$  **and**  $\text{inv } S$   
**and**  $\text{no-dup } (\text{trail } S)$   
**shows**  $\text{no-dup } (\text{trail } T)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-trail-clauses-bound}$ :

**assumes**  
 $\text{cdcl}: \text{cdcl}_{NOT}^{**} S T$  **and**  
 $\text{inv}: \text{inv } S$  **and**  
 $n\text{-d}: \text{no-dup } (\text{trail } S)$  **and**  
 $\text{atms-clauses-}S: \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq A$  **and**

*atms-trail-S*:  $\text{atm-of } \text{'(lits-of-l (trail S))} \subseteq A$   
**shows**  $\text{atm-of } \text{'(lits-of-l (trail T))} \subseteq A \wedge \text{atms-of-mm (clauses}_{NOT} T) \subseteq A$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-all-decomposition-implies*:  
**assumes**  $\text{cdcl}_{NOT}^{**} S T$  **and**  $\text{inv } S$  **and**  $\text{no-dup (trail } S)$  **and**  
 $\text{all-decomposition-implies-m (clauses}_{NOT} S) (\text{get-all-ann-decomposition (trail } S))$   
**shows**  
 $\text{all-decomposition-implies-m (clauses}_{NOT} T) (\text{get-all-ann-decomposition (trail } T))$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-bj-sat-ext-iff*:  
**assumes**  $\text{cdcl}_{NOT}^{**} S T$  **and**  $\text{inv } S$  **and**  $\text{no-dup (trail } S)$   
**shows**  $I \models_{\text{sextm}} \text{clauses}_{NOT} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} T$   
 $\langle \text{proof} \rangle$

**definition** *cdcl<sub>NOT</sub>-NOT-all-inv where*  
 $\text{cdcl}_{NOT}\text{-NOT-all-inv } A S \longleftrightarrow (\text{finite } A \wedge \text{inv } S \wedge \text{atms-of-mm (clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{atm-of } \text{'lits-of-l (trail } S) \subseteq \text{atms-of-ms } A \wedge \text{no-dup (trail } S))$

**lemma** *cdcl<sub>NOT</sub>-NOT-all-inv*:  
**assumes**  $\text{cdcl}_{NOT}^{**} S T$  **and**  $\text{cdcl}_{NOT}\text{-NOT-all-inv } A S$   
**shows**  $\text{cdcl}_{NOT}\text{-NOT-all-inv } A T$   
 $\langle \text{proof} \rangle$

**abbreviation** *learn-or-forget where*  
 $\text{learn-or-forget } S T \equiv \text{learn } S T \vee \text{forget}_{NOT} S T$

**lemma** *rtrancpl-learn-or-forget-cdcl<sub>NOT</sub>*:  
 $\text{learn-or-forget}^{**} S T \implies \text{cdcl}_{NOT}^{**} S T$   
 $\langle \text{proof} \rangle$

**lemma** *learn-or-forget-dpll- $\mu_C$* :  
**assumes**  
 $l\text{-f: learn-or-forget}^{**} S T$  **and**  
 $dpll: dpll\text{-bj } T U$  **and**  
 $\text{inv: cdcl}_{NOT}\text{-NOT-all-inv } A S$   
**shows**  $(2 + \text{card (atms-of-ms } A)) \wedge (1 + \text{card (atms-of-ms } A))$   
 $- \mu_C (1 + \text{card (atms-of-ms } A)) (2 + \text{card (atms-of-ms } A)) (\text{trail-weight } U)$   
 $< (2 + \text{card (atms-of-ms } A)) \wedge (1 + \text{card (atms-of-ms } A))$   
 $- \mu_C (1 + \text{card (atms-of-ms } A)) (2 + \text{card (atms-of-ms } A)) (\text{trail-weight } S)$   
 $(\text{is } ?\mu U < ?\mu S)$   
 $\langle \text{proof} \rangle$

**lemma** *infinite-cdcl<sub>NOT</sub>-exists-learn-and-forget-infinite-chain*:  
**assumes**  
 $\bigwedge i. \text{cdcl}_{NOT} (f i) (f (\text{Suc } i))$  **and**  
 $\text{inv: cdcl}_{NOT}\text{-NOT-all-inv } A (f 0)$   
**shows**  $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i))$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdcl<sub>NOT</sub>-no-learn-and-forget-infinite-chain*:  
**assumes**  
 $\text{no-infinite-lf: } \bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i)))$   
**shows**  $\text{wf } \{(T, S). \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A S\}$

(**is**  $wf \{(T, S). \text{cdcl}_{NOT} S T \wedge ?inv S\}$ )  
 <proof>

**lemma** *inv-and-tranclp-cdcl<sub>NOT</sub>-tranclp-cdcl<sub>NOT</sub>-and-inv:*

$\text{cdcl}_{NOT}^{++} S T \wedge \text{cdcl}_{NOT-NOT-all-inv} A S \longleftrightarrow (\lambda S T. \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT-NOT-all-inv} A S)^{++} S T$

(**is**  $?A \wedge ?I \longleftrightarrow ?B$ )

<proof>

**lemma** *wf-tranclp-cdcl<sub>NOT</sub>-no-learn-and-forget-infinite-chain:*

**assumes**

*no-infinite-lf*:  $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$

**shows**  $wf \{(T, S). \text{cdcl}_{NOT}^{++} S T \wedge \text{cdcl}_{NOT-NOT-all-inv} A S\}$

<proof>

**lemma** *cdcl<sub>NOT</sub>-final-state:*

**assumes**

*n-s*: *no-step*  $\text{cdcl}_{NOT} S$  **and**

*inv*:  $\text{cdcl}_{NOT-NOT-all-inv} A S$  **and**

*decomp*: *all-decomposition-implies-m* ( $\text{clauses}_{NOT} S$ ) (*get-all-ann-decomposition* ( $\text{trail } S$ ))

**shows** *unsatisfiable* ( $\text{set-mset } (\text{clauses}_{NOT} S)$ )

$\vee (\text{trail } S \models_{asm} \text{clauses}_{NOT} S \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} S)))$

<proof>

**lemma** *full-cdcl<sub>NOT</sub>-final-state:*

**assumes**

*full*: *full*  $\text{cdcl}_{NOT} S T$  **and**

*inv*:  $\text{cdcl}_{NOT-NOT-all-inv} A S$  **and**

*n-d*: *no-dup* ( $\text{trail } S$ ) **and**

*decomp*: *all-decomposition-implies-m* ( $\text{clauses}_{NOT} S$ ) (*get-all-ann-decomposition* ( $\text{trail } S$ ))

**shows** *unsatisfiable* ( $\text{set-mset } (\text{clauses}_{NOT} T)$ )

$\vee (\text{trail } T \models_{asm} \text{clauses}_{NOT} T \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} T)))$

<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*  $\text{trail } \text{clauses}_{NOT} \text{ prepend-trail } \text{tl-trail } \text{add-cl}_S \text{ remove-cl}_S +$   
*conflict-driven-clause-learning*  $\text{trail } \text{clauses}_{NOT} \text{ prepend-trail } \text{tl-trail } \text{add-cl}_S \text{ remove-cl}_S$   
*inv* *backjump-conds* *propagate-conds*

$\lambda C S. \text{distinct-mset } C \wedge \neg \text{tautology } C \wedge \text{learn-restrictions } C S \wedge$

$(\exists F K d F' C' L. \text{trail } S = F' @ \text{Decided } K \# F \wedge C = C' + \{\#L\} \wedge F \models_{as} C \text{Not } C'$   
 $\wedge C' + \{\#L\} \notin \text{clauses}_{NOT} S)$

$\lambda C S. \neg (\exists F' F K d L. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{as} C \text{Not } (\text{remove1-mset } L C))$   
 $\wedge \text{forget-restrictions } C S$

**for**  
*trail* :: 'st  $\Rightarrow$  ('v, unit) ann-lits **and**  
*clauses<sub>NOT</sub>* :: 'st  $\Rightarrow$  'v clauses **and**  
*prepend-trail* :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-cl<sub>s</sub><sub>NOT</sub>* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-cl<sub>s</sub><sub>NOT</sub>* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*inv* :: 'st  $\Rightarrow$  bool **and**  
*backjump-conds* :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*propagate-conds* :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*learn-restrictions forget-restrictions* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool  
**begin**

**lemma** *cdcl<sub>NOT</sub>-learn-all-induct*[consumes 1, case-names *dpll-bj learn forget<sub>NOT</sub>*]:  
**fixes** *S T* :: 'st  
**assumes** *cdcl<sub>NOT</sub> S T* **and**  
*dpll*:  $\bigwedge T. \text{dpll-bj } S \ T \Longrightarrow P \ S \ T$  **and**  
*learning*:  
 $\bigwedge C \ F \ K \ F' \ C' \ L \ T. \text{clauses}_{NOT} \ S \models_{pm} C \Longrightarrow$   
 $\text{atms-of } C \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \Longrightarrow$   
 $\text{distinct-mset } C \Longrightarrow$   
 $\neg \text{tautology } C \Longrightarrow$   
 $\text{learn-restrictions } C \ S \Longrightarrow$   
 $\text{trail } S = F' @ \text{Decided } K \ \# \ F \Longrightarrow$   
 $C = C' + \{\#L\# \} \Longrightarrow$   
 $F \models_{as} CNot \ C' \Longrightarrow$   
 $C' + \{\#L\# \} \notin \# \text{clauses}_{NOT} \ S \Longrightarrow$   
 $T \sim \text{add-cl}_{NOT} \ C \ S \Longrightarrow$   
 $P \ S \ T$  **and**  
*forgetting*:  $\bigwedge C \ T. \text{removeAll-mset } C \ (\text{clauses}_{NOT} \ S) \models_{pm} C \Longrightarrow$   
 $C \in \# \text{clauses}_{NOT} \ S \Longrightarrow$   
 $\neg (\exists F' \ F \ K \ L. \text{trail } S = F' @ \text{Decided } K \ \# \ F \wedge F \models_{as} CNot \ (C - \{\#L\# \})) \Longrightarrow$   
 $T \sim \text{remove-cl}_{NOT} \ C \ S \Longrightarrow$   
 $\text{forget-restrictions } C \ S \Longrightarrow$   
 $P \ S \ T$   
**shows**  $P \ S \ T$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-cdcl<sub>NOT</sub>-inv*:  
 $\text{cdcl}_{NOT}^{**} \ S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$   
 $\langle \text{proof} \rangle$

**lemma** *learn-always-simple-clauses*:  
**assumes**  
*learn*:  $\text{learn } S \ T$  **and**  
*n-d*:  $\text{no-dup } (\text{trail } S)$   
**shows**  $\text{set-mset } (\text{clauses}_{NOT} \ T - \text{clauses}_{NOT} \ S)$   
 $\subseteq \text{simple-clss } (\text{atms-of-mm } (\text{clauses}_{NOT} \ S) \cup \text{atm-of } ' \text{ lits-of-l } (\text{trail } S))$   
 $\langle \text{proof} \rangle$

**definition** *conflicting-bj-clss*  $S \equiv$   
 $\{C + \{\#L\# \} \mid C \ L. C + \{\#L\# \} \in \# \text{clauses}_{NOT} \ S \wedge \text{distinct-mset } (C + \{\#L\# \})$   
 $\wedge \neg \text{tautology } (C + \{\#L\# \})$   
 $\wedge (\exists F' \ K \ F. \text{trail } S = F' @ \text{Decided } K \ \# \ F \wedge F \models_{as} CNot \ C)\}$

**lemma** *conflicting-bj-clss-remove-cl<sub>s</sub><sub>NOT</sub>*[simp]:



*conflicting-bj-clss* (*remove-clss*<sub>NOT</sub> *C S*) = *conflicting-bj-clss* *S* - {*C*}  
 ⟨proof⟩

**lemma** *conflicting-bj-clss-remove-clss*<sub>NOT</sub>'[simp]:

*T* ~ *remove-clss*<sub>NOT</sub> *C S*  $\implies$  *conflicting-bj-clss* *T* = *conflicting-bj-clss* *S* - {*C*}  
 ⟨proof⟩

**lemma** *conflicting-bj-clss-add-clss*<sub>NOT</sub>-state-eq:

**assumes**

*T*: *T* ~ *add-clss*<sub>NOT</sub> *C' S* **and**

*n-d*: *no-dup* (*trail S*)

**shows** *conflicting-bj-clss* *T*

= *conflicting-bj-clss* *S*

$\cup$  (*if*  $\exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$

$\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{\text{as}} C \text{Not } C)$

*then* {*C'*} *else* {})

⟨proof⟩

**lemma** *conflicting-bj-clss-add-clss*<sub>NOT</sub>:

*no-dup* (*trail S*)  $\implies$

*conflicting-bj-clss* (*add-clss*<sub>NOT</sub> *C' S*)

= *conflicting-bj-clss* *S*

$\cup$  (*if*  $\exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$

$\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{\text{as}} C \text{Not } C)$

*then* {*C'*} *else* {})

⟨proof⟩

**lemma** *conflicting-bj-clss-incl-clauses*:

*conflicting-bj-clss* *S*  $\subseteq$  *set-mset* (*clauses*<sub>NOT</sub> *S*)

⟨proof⟩

**lemma** *finite-conflicting-bj-clss*[simp]:

*finite* (*conflicting-bj-clss* *S*)

⟨proof⟩

**lemma** *learn-conflicting-increasing*:

*no-dup* (*trail S*)  $\implies$  *learn* *S T*  $\implies$  *conflicting-bj-clss* *S*  $\subseteq$  *conflicting-bj-clss* *T*

⟨proof⟩

**abbreviation** *conflicting-bj-clss-yet* *b S*  $\equiv$

$\mathcal{S} \hat{\wedge} b - \text{card } (\text{conflicting-bj-clss } S)$

**abbreviation**  $\mu_L :: \text{nat} \Rightarrow 'st \Rightarrow \text{nat} \times \text{nat}$  **where**

$\mu_L b S \equiv (\text{conflicting-bj-clss-yet } b S, \text{card } (\text{set-mset } (\text{clauses}_{\text{NOT}} S)))$

**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*<sub>NOT</sub> *S T*

**shows** *conflicting-bj-clss* *S* = *conflicting-bj-clss* *T*

⟨proof⟩

**lemma** *forget- $\mu_L$ -decrease*:

**assumes** *forget*<sub>NOT</sub>: *forget*<sub>NOT</sub> *S T*

**shows**  $(\mu_L \ b \ T, \mu_L \ b \ S) \in \text{less-than } <*lex*> \text{ less-than}$   
 $\langle \text{proof} \rangle$

**lemma** *set-condition-or-split*:

$\{a. (a = b \vee Q \ a) \wedge S \ a\} = (\text{if } S \ b \text{ then } \{b\} \text{ else } \{\}) \cup \{a. Q \ a \wedge S \ a\}$   
 $\langle \text{proof} \rangle$

**lemma** *set-insert-neg*:

$A \neq \text{insert } a \ A \longleftrightarrow a \notin A$   
 $\langle \text{proof} \rangle$

**lemma** *learn- $\mu_L$ -decrease*:

**assumes** *learnST*: *learn*  $S \ T$  **and** *n-d*: *no-dup* (*trail*  $S$ ) **and**  
*A*: *atms-of-mm* (*clauses*<sub>NOT</sub>  $S$ )  $\cup$  *atm-of* ' *lits-of-l* (*trail*  $S$ )  $\subseteq A$  **and**  
*fin-A*: *finite*  $A$   
**shows**  $(\mu_L \ (\text{card } A) \ T, \mu_L \ (\text{card } A) \ S) \in \text{less-than } <*lex*> \text{ less-than}$   
 $\langle \text{proof} \rangle$

We have to assume the following:

- *inv*  $S$ : the invariant holds in the initial 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*<sub>NOT</sub>  $S$ )  $\subseteq$  *atms-of-ms*  $A$ . This can be 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**  $\mu_{CDCL}$  **where**

$\mu_{CDCL} \ A \ T \equiv ((2 + \text{card} \ (\text{atms-of-ms } A)) \wedge (1 + \text{card} \ (\text{atms-of-ms } A))$   
 $\quad - \mu_C \ (1 + \text{card} \ (\text{atms-of-ms } A)) \ (2 + \text{card} \ (\text{atms-of-ms } A)) \ (\text{trail-weight } T),$   
 $\quad \text{conflicting-bj-clss-yet} \ (\text{card} \ (\text{atms-of-ms } A)) \ T, \text{card} \ (\text{set-mset} \ (\text{clauses}_{NOT} \ T)))$

**lemma** *cdcl<sub>NOT</sub>-decreasing-measure*:

**assumes**  
*cdcl<sub>NOT</sub>*  $S \ T$  **and**  
*inv*: *inv*  $S$  **and**  
*atm-clss*: *atms-of-mm* (*clauses*<sub>NOT</sub>  $S$ )  $\subseteq$  *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**  $(\mu_{CDCL} \ A \ T, \mu_{CDCL} \ A \ S)$   
 $\in \text{less-than } <*lex*> \ (\text{less-than } <*lex*> \text{ less-than})$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdcl<sub>NOT</sub>-restricted-learning*:

**assumes** *finite*  $A$   
**shows** *wf*  $\{(T, S).$   
 $(\text{atms-of-mm} \ (\text{clauses}_{NOT} \ S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l} \ (\text{trail } S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{no-dup} \ (\text{trail } S)$   
 $\wedge \text{inv } S)$   
 $\wedge \text{cdcl}_{NOT} \ S \ T \}$   
 $\langle \text{proof} \rangle$

**definition**  $\mu_C' :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow \text{nat}$  **where**

$\mu_C' \ A \ T \equiv \mu_C \ (1 + \text{card} \ (\text{atms-of-ms } A)) \ (2 + \text{card} \ (\text{atms-of-ms } A)) \ (\text{trail-weight } T)$

**definition**  $\mu_{CDCL}' :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow nat$  **where**

$\mu_{CDCL}' A T \equiv$   
 $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T) * (1 + 3^{\text{card } (\text{atms-of-ms } A)}) * 2$   
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T * 2$   
 $+ \text{card } (\text{set-mset } (\text{clauses}_{NOT} T))$

**lemma**  $\text{cdcl}_{NOT}\text{-decreasing-measure}'$ :

**assumes**

$\text{cdcl}_{NOT} S T$  **and**

$\text{inv } S$  **and**

$\text{atms-clss: atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$  **and**

$\text{atms-trail: atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A$  **and**

$n\text{-d: no-dup } (\text{trail } S)$  **and**

$\text{fin-A: finite } A$

**shows**  $\mu_{CDCL}' A T < \mu_{CDCL}' A S$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_{NOT}\text{-clauses-bound}$ :

**assumes**

$\text{cdcl}_{NOT} S T$  **and**

$\text{inv } S$  **and**

$\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq A$  **and**

$\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq A$  **and**

$n\text{-d: no-dup } (\text{trail } S)$  **and**

$\text{fin-A[simp]: finite } A$

**shows**  $\text{set-mset } (\text{clauses}_{NOT} T) \subseteq \text{set-mset } (\text{clauses}_{NOT} S) \cup \text{simple-clss } A$

$\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-clauses-bound}$ :

**assumes**

$\text{cdcl}_{NOT}^{**} S T$  **and**

$\text{inv } S$  **and**

$\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq A$  **and**

$\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq A$  **and**

$n\text{-d: no-dup } (\text{trail } S)$  **and**

$\text{finite: finite } A$

**shows**  $\text{set-mset } (\text{clauses}_{NOT} T) \subseteq \text{set-mset } (\text{clauses}_{NOT} S) \cup \text{simple-clss } A$

$\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-card-clauses-bound}$ :

**assumes**

$\text{cdcl}_{NOT}^{**} S T$  **and**

$\text{inv } S$  **and**

$\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq A$  **and**

$\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq A$  **and**

$n\text{-d: no-dup } (\text{trail } S)$  **and**

$\text{finite: finite } A$

**shows**  $\text{card } (\text{set-mset } (\text{clauses}_{NOT} T)) \leq \text{card } (\text{set-mset } (\text{clauses}_{NOT} S)) + 3 \wedge (\text{card } A)$

$\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-card-clauses-bound}'$ :

**assumes**

$\text{cdcl}_{NOT}^{**} S T$  **and**

$\text{inv } S$  **and**

$atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$  **and**  
 $atm\text{-}of\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$  **and**  
 $n\text{-}d\text{:}\ no\text{-}dup\ (trail\ S)$  **and**  
 $finite\text{:}\ finite\ A$   
**shows**  $card\ \{C \mid C.\ C \in \# clauses_{NOT}\ T \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\}$   
 $\leq card\ \{C \mid C.\ C \in \# clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3 \wedge (card\ A)$   
 $(is\ card\ ?T \leq card\ ?S + -)$   
 $\langle proof \rangle$

**lemma**  $rtranclp\text{-}cdcl_{NOT}\text{-}card\text{-}simple\text{-}clauses\text{-}bound$ :

**assumes**  
 $cdcl_{NOT}^{**}\ S\ T$  **and**  
 $inv\ S$  **and**  
 $NA\text{:}\ atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$  **and**  
 $MA\text{:}\ atm\text{-}of\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$  **and**  
 $n\text{-}d\text{:}\ no\text{-}dup\ (trail\ S)$  **and**  
 $finite\text{:}\ finite\ A$   
**shows**  $card\ (set\text{-}mset\ (clauses_{NOT}\ T))$   
 $\leq card\ \{C.\ C \in \# clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3 \wedge (card\ A)$   
 $(is\ card\ ?T \leq card\ ?S + -)$   
 $\langle proof \rangle$

**definition**  $\mu_{CDCL}'\text{-}bound :: 'v\ clause\ set \Rightarrow 'st \Rightarrow nat$  **where**

$\mu_{CDCL}'\text{-}bound\ A\ S =$   
 $((2 + card\ (atms\text{-}of\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-}of\text{-}ms\ A))) * (1 + 3 \wedge card\ (atms\text{-}of\text{-}ms\ A)) * 2$   
 $+ 2 * 3 \wedge (card\ (atms\text{-}of\text{-}ms\ A))$   
 $+ card\ \{C.\ C \in \# clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3 \wedge (card\ (atms\text{-}of\text{-}ms\ A))$

**lemma**  $\mu_{CDCL}'\text{-}bound\text{-}reduce\text{-}trail\text{-}to_{NOT}[simp]$ :

$\mu_{CDCL}'\text{-}bound\ A\ (reduce\text{-}trail\text{-}to_{NOT}\ M\ S) = \mu_{CDCL}'\text{-}bound\ A\ S$   
 $\langle proof \rangle$

**lemma**  $rtranclp\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-}bound\text{-}reduce\text{-}trail\text{-}to_{NOT}$ :

**assumes**  
 $cdcl_{NOT}^{**}\ S\ T$  **and**  
 $inv\ S$  **and**  
 $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $atm\text{-}of\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $n\text{-}d\text{:}\ no\text{-}dup\ (trail\ S)$  **and**  
 $finite\text{:}\ finite\ (atms\text{-}of\text{-}ms\ A)$  **and**  
 $U\text{:}\ U \sim reduce\text{-}trail\text{-}to_{NOT}\ M\ T$   
**shows**  $\mu_{CDCL}'\ A\ U \leq \mu_{CDCL}'\text{-}bound\ A\ S$   
 $\langle proof \rangle$

**lemma**  $rtranclp\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-}bound$ :

**assumes**  
 $cdcl_{NOT}^{**}\ S\ T$  **and**  
 $inv\ S$  **and**  
 $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $atm\text{-}of\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}ms\ A$  **and**  
 $n\text{-}d\text{:}\ no\text{-}dup\ (trail\ S)$  **and**  
 $finite\text{:}\ finite\ (atms\text{-}of\text{-}ms\ A)$   
**shows**  $\mu_{CDCL}'\ A\ T \leq \mu_{CDCL}'\text{-}bound\ A\ S$   
 $\langle proof \rangle$

**lemma**  $rtranclp\text{-}\mu_{CDCL}'\text{-}bound\text{-}decreasing$ :

```

assumes
   $cdcl_{NOT}^{**} S T$  and
   $inv S$  and
   $atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A$  and
   $atm-of (lits-of-l (trail S)) \subseteq atms-of-ms A$  and
   $n-d: no-dup (trail S)$  and
   $finite[simp]: finite (atms-of-ms A)$ 
shows  $\mu_{CDCL}'\text{-bound } A T \leq \mu_{CDCL}'\text{-bound } A S$ 
 $\langle proof \rangle$ 

end — end of conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt

```

### 5.2.5 CDCL with restarts

#### Definition

```

locale restart-ops =
  fixes
     $cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$  and
     $restart :: 'st \Rightarrow 'st \Rightarrow bool$ 
  begin
    inductive  $cdcl_{NOT}\text{-raw-restart} :: 'st \Rightarrow 'st \Rightarrow bool$  where
       $cdcl_{NOT} S T \Longrightarrow cdcl_{NOT}\text{-raw-restart } S T \mid$ 
       $restart S T \Longrightarrow cdcl_{NOT}\text{-raw-restart } S T$ 
  end

locale conflict-driven-clause-learning-with-restarts =
  conflict-driven-clause-learning trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds learn-cond forget-cond
  for
     $trail :: 'st \Rightarrow ('v, unit) \text{ ann-lits}$  and
     $clauses_{NOT} :: 'st \Rightarrow 'v \text{ clauses}$  and
     $prepend-trail :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow 'st$  and
     $tl-trail :: 'st \Rightarrow 'st$  and
     $add-cl_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$  and
     $remove-cl_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$  and
     $inv :: 'st \Rightarrow bool$  and
     $backjump-conds :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$  and
     $propagate-conds :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow bool$  and
     $learn-cond \text{ forget-cond} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow bool$ 
  begin

lemma  $cdcl_{NOT}\text{-iff-}cdcl_{NOT}\text{-raw-restart-no-restarts}$ :
   $cdcl_{NOT} S T \longleftrightarrow restart\text{-ops}.cdcl_{NOT}\text{-raw-restart } cdcl_{NOT} (\lambda -. False) S T$ 
  (is ?C S T  $\longleftrightarrow$  ?R S T)
 $\langle proof \rangle$ 

lemma  $cdcl_{NOT}\text{-}cdcl_{NOT}\text{-raw-restart}$ :
   $cdcl_{NOT} S T \Longrightarrow restart\text{-ops}.cdcl_{NOT}\text{-raw-restart } cdcl_{NOT} restart S T$ 
 $\langle proof \rangle$ 
end

```

#### Increasing restarts

To add restarts we need 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  $\mu$ : it should decrease under the assumptions  $bound\_inv$ , whenever a  $cdcl_{NOT}$  or a  $restart$  is done. A parameter is given to  $\mu$ : 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  $cdcl_{NOT}\text{-inv}$  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  $\mu\text{-bound}$  taking the same parameter as  $\mu$  and the initial state of the considered  $cdcl_{NOT}$  chain.

```

locale  $cdcl_{NOT}\text{-increasing-restarts-ops}$  =
  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
   $\mu$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat and
   $cdcl_{NOT}\text{-inv}$  :: 'st  $\Rightarrow$  bool and
   $\mu\text{-bound}$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat
assumes
  f: unbounded f and
  f-ge-1:  $\bigwedge n. n \geq 1 \Rightarrow f\ n \neq 0$  and
  bound-inv:  $\bigwedge A\ S\ T. cdcl_{NOT}\text{-inv}\ S \Rightarrow bound\_inv\ A\ S \Rightarrow cdcl_{NOT}\ S\ T \Rightarrow bound\_inv\ A\ T$  and
   $cdcl_{NOT}\text{-measure}$ :  $\bigwedge A\ S\ T. cdcl_{NOT}\text{-inv}\ S \Rightarrow bound\_inv\ A\ S \Rightarrow cdcl_{NOT}\ S\ T \Rightarrow \mu\ A\ T < \mu$ 
  A S and
  measure-bound2:  $\bigwedge A\ T\ U. cdcl_{NOT}\text{-inv}\ T \Rightarrow bound\_inv\ A\ T \Rightarrow cdcl_{NOT}^{**}\ T\ U$ 
     $\Rightarrow \mu\ A\ U \leq \mu\text{-bound}\ A\ T$  and
  measure-bound4:  $\bigwedge A\ T\ U. cdcl_{NOT}\text{-inv}\ T \Rightarrow bound\_inv\ A\ T \Rightarrow cdcl_{NOT}^{**}\ T\ U$ 
     $\Rightarrow \mu\text{-bound}\ A\ U \leq \mu\text{-bound}\ A\ T$  and
   $cdcl_{NOT}\text{-restart-inv}$ :  $\bigwedge A\ U\ V. cdcl_{NOT}\text{-inv}\ U \Rightarrow restart\ U\ V \Rightarrow bound\_inv\ A\ U \Rightarrow bound\_inv$ 
  A V
and
  exists-bound:  $\bigwedge R\ S. cdcl_{NOT}\text{-inv}\ R \Rightarrow restart\ R\ S \Rightarrow \exists A. bound\_inv\ A\ S$  and
   $cdcl_{NOT}\text{-inv}$ :  $\bigwedge S\ T. cdcl_{NOT}\text{-inv}\ S \Rightarrow cdcl_{NOT}\ S\ T \Rightarrow cdcl_{NOT}\text{-inv}\ T$  and
   $cdcl_{NOT}\text{-inv-restart}$ :  $\bigwedge S\ T. cdcl_{NOT}\text{-inv}\ S \Rightarrow restart\ S\ T \Rightarrow cdcl_{NOT}\text{-inv}\ T$ 
begin

```

**lemma**  $cdcl_{NOT}\text{-}cdcl_{NOT}\text{-inv}$ :

```

assumes
  ( $cdcl_{NOT} \rightsquigarrow n$ ) S T and
   $cdcl_{NOT}\text{-inv}\ S$ 
shows  $cdcl_{NOT}\text{-inv}\ T$ 
<proof>

```

**lemma**  $cdcl_{NOT}\text{-bound-inv}$ :

```

assumes

```

$(cdcl_{NOT} \rightsquigarrow n) S T$  **and**  
 $cdcl_{NOT-inv} S$   
 $bound-inv A S$   
**shows**  $bound-inv A T$   
 $\langle proof \rangle$

**lemma**  $rtrancplp-cdcl_{NOT}-cdcl_{NOT-inv}$ :

**assumes**  
 $cdcl_{NOT}^{**} S T$  **and**  
 $cdcl_{NOT-inv} S$   
**shows**  $cdcl_{NOT-inv} T$   
 $\langle proof \rangle$

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

**assumes**  
 $cdcl_{NOT}^{**} S T$  **and**  
 $bound-inv A S$  **and**  
 $cdcl_{NOT-inv} S$   
**shows**  $bound-inv A T$   
 $\langle proof \rangle$

**lemma**  $cdcl_{NOT}-comp-n-le$ :

**assumes**  
 $(cdcl_{NOT} \rightsquigarrow (Suc\ n)) S T$  **and**  
 $bound-inv A S$   
 $cdcl_{NOT-inv} S$   
**shows**  $\mu A T < \mu A S - n$   
 $\langle proof \rangle$

**lemma**  $wf-cdcl_{NOT}$ :

$wf \{(T, S). cdcl_{NOT} S T \wedge cdcl_{NOT-inv} S \wedge bound-inv A S\}$  (**is**  $wf\ ?A$ )  
 $\langle proof \rangle$

**lemma**  $rtrancplp-cdcl_{NOT}-measure$ :

**assumes**  
 $cdcl_{NOT}^{**} S T$  **and**  
 $bound-inv A S$  **and**  
 $cdcl_{NOT-inv} S$   
**shows**  $\mu A T \leq \mu A S$   
 $\langle proof \rangle$

**lemma**  $cdcl_{NOT}-comp-bounded$ :

**assumes**  
 $bound-inv A S$  **and**  $cdcl_{NOT-inv} S$  **and**  $m \geq 1 + \mu A S$   
**shows**  $\neg(cdcl_{NOT} \rightsquigarrow m) S T$   
 $\langle proof \rangle$

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

**inductive**  $cdcl_{NOT}-restart$  **where**

$restart-step: (cdcl_{NOT} \rightsquigarrow m) S T \implies m \geq f\ n \implies restart\ T\ U$   
 $\implies cdcl_{NOT}-restart\ (S, n)\ (U, Suc\ n) \mid$

$restart-full: full1\ cdcl_{NOT} S T \implies cdcl_{NOT}-restart\ (S, n)\ (T, Suc\ n)$

**lemmas**  $cdcl_{NOT}-with-restart-induct = cdcl_{NOT}-restart.induct[split-format(complete),$

*OF cdcl<sub>NOT</sub>-increasing-restarts-ops-axioms]*

**lemma** *cdcl<sub>NOT</sub>-restart-cdcl<sub>NOT</sub>-raw-restart:*

*cdcl<sub>NOT</sub>-restart S T  $\implies$  cdcl<sub>NOT</sub>-raw-restart\*\* (fst S) (fst T)*  
*<proof>*

**lemma** *cdcl<sub>NOT</sub>-with-restart-bound-inv:*

**assumes**  
*cdcl<sub>NOT</sub>-restart S T and*  
*bound-inv A (fst S) and*  
*cdcl<sub>NOT</sub>-inv (fst S)*  
**shows** *bound-inv A (fst T)*  
*<proof>*

**lemma** *cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv:*

**assumes**  
*cdcl<sub>NOT</sub>-restart S T and*  
*cdcl<sub>NOT</sub>-inv (fst S)*  
**shows** *cdcl<sub>NOT</sub>-inv (fst T)*  
*<proof>*

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv:*

**assumes**  
*cdcl<sub>NOT</sub>-restart\*\* S T and*  
*cdcl<sub>NOT</sub>-inv (fst S)*  
**shows** *cdcl<sub>NOT</sub>-inv (fst T)*  
*<proof>*

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-bound-inv:*

**assumes**  
*cdcl<sub>NOT</sub>-restart\*\* S T and*  
*cdcl<sub>NOT</sub>-inv (fst S) and*  
*bound-inv A (fst S)*  
**shows** *bound-inv A (fst T)*  
*<proof>*

**lemma** *cdcl<sub>NOT</sub>-with-restart-increasing-number:*

*cdcl<sub>NOT</sub>-restart S T  $\implies$  snd T = 1 + snd S*  
*<proof>*

**end**

**locale** *cdcl<sub>NOT</sub>-increasing-restarts =*

*cdcl<sub>NOT</sub>-increasing-restarts-ops restart cdcl<sub>NOT</sub> f bound-inv  $\mu$  cdcl<sub>NOT</sub>-inv  $\mu$ -bound +*  
*dpll-state trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub>*

**for**

*trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and*  
*clauses<sub>NOT</sub> :: 'st  $\Rightarrow$  'v clauses and*  
*prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*tl-trail :: 'st  $\Rightarrow$  'st and*  
*add-cl<sub>NOT</sub> :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*remove-cl<sub>NOT</sub> :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*f :: nat  $\Rightarrow$  nat and*  
*restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and*  
*bound-inv :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  bool and*  
 *$\mu$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat and*  
*cdcl<sub>NOT</sub> :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and*



$cdcl_{NOT-inv} :: 'st \Rightarrow bool$  **and**  
 $\mu-bound :: 'bound \Rightarrow 'st \Rightarrow nat +$   
**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**  
 $cdcl_{NOT-raw-restart-\mu-bound}:$   
 $cdcl_{NOT-restart}\ (T, a)\ (V, b) \Longrightarrow cdcl_{NOT-inv}\ T \Longrightarrow bound-inv\ A\ T$   
 $\Longrightarrow \mu-bound\ A\ V \leq \mu-bound\ A\ T$   
**begin**

**lemma**  $rtrancpl-cdcl_{NOT-raw-restart-\mu-bound}:$   
 $cdcl_{NOT-restart}^{**}\ (T, a)\ (V, b) \Longrightarrow cdcl_{NOT-inv}\ T \Longrightarrow bound-inv\ A\ T$   
 $\Longrightarrow \mu-bound\ A\ V \leq \mu-bound\ A\ T$   
 $\langle proof \rangle$

**lemma**  $cdcl_{NOT-raw-restart-measure-bound}:$   
 $cdcl_{NOT-restart}\ (T, a)\ (V, b) \Longrightarrow cdcl_{NOT-inv}\ T \Longrightarrow bound-inv\ A\ T$   
 $\Longrightarrow \mu\ A\ V \leq \mu-bound\ A\ T$   
 $\langle proof \rangle$

**lemma**  $rtrancpl-cdcl_{NOT-raw-restart-measure-bound}:$   
 $cdcl_{NOT-restart}^{**}\ (T, a)\ (V, b) \Longrightarrow cdcl_{NOT-inv}\ T \Longrightarrow bound-inv\ A\ T$   
 $\Longrightarrow \mu\ A\ V \leq \mu-bound\ A\ T$   
 $\langle proof \rangle$

**lemma**  $wf-cdcl_{NOT-restart}:$   
 $wf\ \{(T, S). cdcl_{NOT-restart}\ S\ T \wedge cdcl_{NOT-inv}\ (fst\ S)\}$  **(is**  $wf\ ?A)$   
 $\langle proof \rangle$

**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**  $full1\ cdcl_{NOT}\ (fst\ S)\ (fst\ T)$   
 $\langle proof \rangle$

**lemma**  $rtrancpl-cdcl_{NOT-with-inv-inv-rtrancpl-cdcl_{NOT}}:$   
**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**  $?A^{**}\ S\ T \longleftrightarrow ?B^{**}\ S\ T)$   
 $\langle proof \rangle$

**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**  $no-step\ cdcl_{NOT}\ (fst\ S)$   
 $\langle proof \rangle$

**end**

### 5.2.6 Merging backjump and learning

**locale** *cdcl<sub>NOT</sub>-merge-bj-learn-ops* =  
*decide-ops trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> +*  
*forget-ops trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> forget-cond +*  
*propagate-ops trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> propagate-conds*  
**for**  
*trail :: 'st ⇒ ('v, unit) ann-lits and*  
*clauses<sub>NOT</sub> :: 'st ⇒ 'v clauses and*  
*prepend-trail :: ('v, unit) ann-lit ⇒ 'st ⇒ 'st and*  
*tl-trail :: 'st ⇒ 'st and*  
*add-cl<sub>NOT</sub> :: 'v clause ⇒ 'st ⇒ 'st and*  
*remove-cl<sub>NOT</sub> :: '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*  
 $\Rightarrow$  *no-dup (trail S)*  
 $\Rightarrow$   $T \sim \text{prepend-trail } (\text{Propagated } L \ ()) \ (\text{reduce-trail-to}_{NOT} \ F \ (\text{add-cl}_{NOT} \ C'' \ S))$   
 $\Rightarrow$   $C \in \# \text{ clauses}_{NOT} \ S$   
 $\Rightarrow$   $\text{trail } S \models_{as} CNot \ C$   
 $\Rightarrow$  *undefined-lit F L*  
 $\Rightarrow$   $\text{atm-of } L \in \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \cup \text{atm-of ' (lits-of-l (trail S))}$   
 $\Rightarrow$   $\text{clauses}_{NOT} \ S \models_{pm} C' + \{\#L\# \}$   
 $\Rightarrow$   $C'' = C' + \{\#L\# \}$   
 $\Rightarrow$   $F \models_{as} CNot \ C'$   
 $\Rightarrow$  *backjump-l-cond C C' L S T*  
 $\Rightarrow$  *backjump-l S T*

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

**declare** *reduce-trail-to<sub>NOT</sub>-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<sub>NOT</sub>-length-ne[simp] Set.Un-iff[simp] Set.insert-iff[simp]*

**inductive** *cdcl<sub>NOT</sub>-merged-bj-learn :: 'st ⇒ 'st ⇒ bool for S :: 'st where*  
*cdcl<sub>NOT</sub>-merged-bj-learn-decide<sub>NOT</sub>: decide<sub>NOT</sub> S S' ⇒ cdcl<sub>NOT</sub>-merged-bj-learn S S' |*  
*cdcl<sub>NOT</sub>-merged-bj-learn-propagate<sub>NOT</sub>: propagate<sub>NOT</sub> S S' ⇒ cdcl<sub>NOT</sub>-merged-bj-learn S S' |*  
*cdcl<sub>NOT</sub>-merged-bj-learn-backjump-l: backjump-l S S' ⇒ cdcl<sub>NOT</sub>-merged-bj-learn S S' |*  
*cdcl<sub>NOT</sub>-merged-bj-learn-forget<sub>NOT</sub>: forget<sub>NOT</sub> S S' ⇒ cdcl<sub>NOT</sub>-merged-bj-learn S S'*

**lemma** *cdcl<sub>NOT</sub>-merged-bj-learn-no-dup-inv:*  
 $\text{cdcl}_{NOT}\text{-merged-bj-learn } S \ T \Rightarrow \text{no-dup } (\text{trail } S) \Rightarrow \text{no-dup } (\text{trail } T)$   
 $\langle \text{proof} \rangle$   
**end**

**locale** *cdcl<sub>NOT</sub>-merge-bj-learn-proxy* =  
*cdcl<sub>NOT</sub>-merge-bj-learn-ops trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub>*  
*propagate-conds forget-cond*  
 $\lambda C \ C' \ L' \ S \ T. \text{backjump-l-cond } C \ C' \ L' \ S \ T$   
 $\wedge \text{distinct-mset } (C' + \{\#L'\# \}) \wedge \neg \text{tautology } (C' + \{\#L'\# \})$

**for**  
*trail* :: 'st  $\Rightarrow$  ('v, unit) ann-lits **and**  
*clauses<sub>NOT</sub>* :: 'st  $\Rightarrow$  'v clauses **and**  
*prepend-trail* :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-cl<sub>s</sub><sub>NOT</sub>* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-cl<sub>s</sub><sub>NOT</sub>* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*propagate-con<sub>s</sub>* :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*forget-cond* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*backjump-l-cond* :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool +  
**fixes**  
*inv* :: 'st  $\Rightarrow$  bool  
**assumes**  
*bj-merge-can-jump*:  
 $\bigwedge S C F' K F L.$   
*inv S*  
 $\Rightarrow$  trail  $S = F' @ Decided K \# F$   
 $\Rightarrow C \in \# clauses_{NOT} S$   
 $\Rightarrow$  trail  $S \models_{as} CNot C$   
 $\Rightarrow$  undefined-lit  $F L$   
 $\Rightarrow atm-of L \in atms-of-mm (clauses_{NOT} S) \cup atm-of ' (lits-of-l (F' @ Decided K \# F))$   
 $\Rightarrow clauses_{NOT} S \models_{pm} C' + \{\#L\#\}$   
 $\Rightarrow F \models_{as} CNot C'$   
 $\Rightarrow \neg no-step backjump-l S$  **and**  
*cdcl-merged-inv*:  $\bigwedge S T. cdcl_{NOT}\text{-merged-bj-learn } S T \Rightarrow inv S \Rightarrow inv T$   
**begin**

**abbreviation** *backjump-con<sub>s</sub>* :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool  
**where**  
*backjump-con<sub>s</sub>*  $\equiv \lambda C C' L' S T. distinct-mset (C' + \{\#L'\#\}) \wedge \neg tautology (C' + \{\#L'\#\})$

Without additional knowledge on *backjump-l-cond*, it is impossible to have the same invariant.

**sublocale** *dpll-with-backjumping-ops* trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>s</sub><sub>NOT</sub> remove-cl<sub>s</sub><sub>NOT</sub>  
*inv backjump-con<sub>s</sub> propagate-con<sub>s</sub>*  
 <proof>

**end**

**locale** *cdcl<sub>NOT</sub>-merge-bj-learn-proxy2* =  
*cdcl<sub>NOT</sub>-merge-bj-learn-proxy* trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>s</sub><sub>NOT</sub> remove-cl<sub>s</sub><sub>NOT</sub>  
*propagate-con<sub>s</sub> forget-cond backjump-l-cond inv*  
**for**  
*trail* :: 'st  $\Rightarrow$  ('v, unit) ann-lits **and**  
*clauses<sub>NOT</sub>* :: 'st  $\Rightarrow$  'v clauses **and**  
*prepend-trail* :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-cl<sub>s</sub><sub>NOT</sub>* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-cl<sub>s</sub><sub>NOT</sub>* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*propagate-con<sub>s</sub>* :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*forget-cond* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*backjump-l-cond* :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
*inv* :: 'st  $\Rightarrow$  bool  
**begin**

**sublocale** *conflict-driven-clause-learning-ops* trail clauses<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>s</sub><sub>NOT</sub>  
*remove-cl<sub>s</sub><sub>NOT</sub> inv backjump-con<sub>s</sub> propagate-con<sub>s</sub>*

$\lambda C \cdot \text{distinct-mset } C \wedge \neg \text{tautology } C$   
 $\text{forget-cond}$   
 $\langle \text{proof} \rangle$   
**end**

**locale**  $\text{cdcl}_{NOT}\text{-merge-bj-learn} =$   
 $\text{cdcl}_{NOT}\text{-merge-bj-learn-proxy2 trail clauses}_{NOT} \text{prepend-trail tl-trail add-cl}_{NOT} \text{remove-cl}_{NOT}$   
 $\text{propagate-conds forget-cond backjump-l-cond inv}$   
**for**  
 $\text{trail} :: 'st \Rightarrow ('v, \text{unit}) \text{ann-lits and}$   
 $\text{clauses}_{NOT} :: 'st \Rightarrow 'v \text{ clauses and}$   
 $\text{prepend-trail} :: ('v, \text{unit}) \text{ann-lit} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $\text{tl-trail} :: 'st \Rightarrow 'st \text{ and}$   
 $\text{add-cl}_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $\text{remove-cl}_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $\text{backjump-l-cond} :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow \text{bool and}$   
 $\text{propagate-conds} :: ('v, \text{unit}) \text{ann-lit} \Rightarrow 'st \Rightarrow \text{bool and}$   
 $\text{forget-cond} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow \text{bool and}$   
 $\text{inv} :: 'st \Rightarrow \text{bool} +$   
**assumes**  
 $\text{dpll-merge-bj-inv: } \bigwedge S T. \text{dpll-bj } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T \text{ and}$   
 $\text{learn-inv: } \bigwedge S T. \text{learn } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$   
**begin**

**sublocale**  
 $\text{conflict-driven-clause-learning trail clauses}_{NOT} \text{prepend-trail tl-trail add-cl}_{NOT} \text{remove-cl}_{NOT}$   
 $\text{inv backjump-conds propagate-conds}$   
 $\lambda C \cdot \text{distinct-mset } C \wedge \neg \text{tautology } C$   
 $\text{forget-cond}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{backjump-l-learn-backjump:}$   
**assumes**  $\text{bt: backjump-l } S T \text{ and inv: inv } S \text{ and n-d: no-dup (trail } S)$   
**shows**  $\exists C' L D. \text{learn } S (\text{add-cl}_{NOT} D S)$   
 $\wedge D = (C' + \{\#L\# \})$   
 $\wedge \text{backjump } (\text{add-cl}_{NOT} D S) T$   
 $\wedge \text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' (\text{lits-of-l (trail } S))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_{NOT}\text{-merged-bj-learn-is-tranclp-cdcl}_{NOT}:$   
 $\text{cdcl}_{NOT}\text{-merged-bj-learn } S T \Longrightarrow \text{inv } S \Longrightarrow \text{no-dup (trail } S) \Longrightarrow \text{cdcl}_{NOT}^{++} S T$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-merged-bj-learn-is-rtranclp-cdcl}_{NOT}\text{-and-inv:}$   
 $\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T \Longrightarrow \text{inv } S \Longrightarrow \text{no-dup (trail } S) \Longrightarrow \text{cdcl}_{NOT}^{**} S T \wedge \text{inv } T$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-merged-bj-learn-is-rtranclp-cdcl}_{NOT}:$   
 $\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T \Longrightarrow \text{inv } S \Longrightarrow \text{no-dup (trail } S) \Longrightarrow \text{cdcl}_{NOT}^{**} S T$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-merged-bj-learn-inv:}$   
 $\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T \Longrightarrow \text{inv } S \Longrightarrow \text{no-dup (trail } S) \Longrightarrow \text{inv } T$   
 $\langle \text{proof} \rangle$

**definition**  $\mu_C' :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow \text{nat where}$

$\mu_C' A T \equiv \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$

**definition**  $\mu_{CDCL}'\text{-merged} :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow \text{nat}$  **where**

$\mu_{CDCL}'\text{-merged } A T \equiv$   
 $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A T) * 2 + \text{card} (\text{set-mset} (\text{clauses}_{NOT} T))$

**lemma**  $\text{cdcl}_{NOT}\text{-decreasing-measure}'$ :

**assumes**

$\text{cdcl}_{NOT}\text{-merged-bj-learn } S T$  **and**

$\text{inv: inv } S$  **and**

$\text{atm-clss: atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$  **and**

$\text{atm-trail: atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$  **and**

$n\text{-d: no-dup } (\text{trail } S)$  **and**

$\text{fin-A: finite } A$

**shows**  $\mu_{CDCL}'\text{-merged } A T < \mu_{CDCL}'\text{-merged } A S$

$\langle \text{proof} \rangle$

**lemma**  $\text{wf-cdcl}_{NOT}\text{-merged-bj-learn}$ :

**assumes**

$\text{fin-A: finite } A$

**shows**  $\text{wf } \{(T, S)\}$ .

$(\text{inv } S \wedge \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{no-dup } (\text{trail } S))$

$\wedge \text{cdcl}_{NOT}\text{-merged-bj-learn } S T\}$

$\langle \text{proof} \rangle$

**lemma**  $\text{trancpl-cdcl}_{NOT}\text{-cdcl}_{NOT}\text{-trancpl}$ :

**assumes**

$\text{cdcl}_{NOT}\text{-merged-bj-learn}^{++} S T$  **and**

$\text{inv: inv } S$  **and**

$\text{atm-clss: atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$  **and**

$\text{atm-trail: atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$  **and**

$n\text{-d: no-dup } (\text{trail } S)$  **and**

$\text{fin-A[simp]: finite } A$

**shows**  $(T, S) \in \{(T, S)\}$ .

$(\text{inv } S \wedge \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{no-dup } (\text{trail } S))$

$\wedge \text{cdcl}_{NOT}\text{-merged-bj-learn } S T\}^+ (\text{is } - \in ?P^+)$

$\langle \text{proof} \rangle$

**lemma**  $\text{wf-trancpl-cdcl}_{NOT}\text{-merged-bj-learn}$ :

**assumes**  $\text{finite } A$

**shows**  $\text{wf } \{(T, S)\}$ .

$(\text{inv } S \wedge \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{no-dup } (\text{trail } S))$

$\wedge \text{cdcl}_{NOT}\text{-merged-bj-learn}^{++} S T\}$

$\langle \text{proof} \rangle$

**lemma**  $\text{backjump-no-step-backjump-l}$ :

$\text{backjump } S T \implies \text{inv } S \implies \neg \text{no-step backjump-l } S$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_{NOT}\text{-merged-bj-learn-final-state}$ :

**fixes**  $A :: 'v \text{ clause set}$  **and**  $S T :: 'st$

**assumes**

*n-s*: no-step  $cdcl_{NOT}$ -merged-bj-learn  $S$  **and**  
*atms-S*:  $atms\text{-of}\text{-mm} (clauses_{NOT} S) \subseteq atms\text{-of}\text{-ms} A$  **and**  
*atms-trail*:  $atm\text{-of} \text{ ' lits-of-l } (trail S) \subseteq atms\text{-of}\text{-ms} A$  **and**  
*n-d*: no-dup  $(trail S)$  **and**  
*finite A* **and**  
*inv*:  $inv S$  **and**  
*decomp*: all-decomposition-implies-m  $(clauses_{NOT} S)$  (get-all-ann-decomposition  $(trail S)$ )  
**shows** unsatisfiable (set-mset  $(clauses_{NOT} S)$ )  
 $\vee (trail S \models_{asm} clauses_{NOT} S \wedge \text{satisfiable} (set\text{-mset} (clauses_{NOT} S)))$   
 <proof>

**lemma** full- $cdcl_{NOT}$ -merged-bj-learn-final-state:

**fixes**  $A :: \text{'v clause set}$  **and**  $S T :: \text{'st}$   
**assumes**  
*full*: full  $cdcl_{NOT}$ -merged-bj-learn  $S T$  **and**  
*atms-S*:  $atms\text{-of}\text{-mm} (clauses_{NOT} S) \subseteq atms\text{-of}\text{-ms} A$  **and**  
*atms-trail*:  $atm\text{-of} \text{ ' lits-of-l } (trail S) \subseteq atms\text{-of}\text{-ms} A$  **and**  
*n-d*: no-dup  $(trail S)$  **and**  
*finite A* **and**  
*inv*:  $inv S$  **and**  
*decomp*: all-decomposition-implies-m  $(clauses_{NOT} S)$  (get-all-ann-decomposition  $(trail S)$ )  
**shows** unsatisfiable (set-mset  $(clauses_{NOT} T)$ )  
 $\vee (trail T \models_{asm} clauses_{NOT} T \wedge \text{satisfiable} (set\text{-mset} (clauses_{NOT} T)))$   
 <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<sub>NOT</sub> prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub>  
 inv backjump-conds propagate-conds learn-restrictions forget-restrictions

**for**

trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits **and**  
 clauses<sub>NOT</sub> :: 'st  $\Rightarrow$  'v clauses **and**  
 prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
 tl-trail :: 'st  $\Rightarrow$  'st **and**  
 add-cl<sub>NOT</sub> :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
 remove-cl<sub>NOT</sub> :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
 inv :: 'st  $\Rightarrow$  bool **and**  
 backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
 propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool **and**  
 learn-restrictions forget-restrictions :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool

+

**fixes**  $f :: nat \Rightarrow nat$

**assumes**

unbounded: unbounded  $f$  **and**  $f\text{-ge-1}: \bigwedge n. n \geq 1 \Rightarrow f n \geq 1$  **and**  
 inv-restart:  $\bigwedge S T. inv S \Rightarrow T \sim \text{reduce-trail-to}_{NOT} ([::\text{'a list}) S \Rightarrow inv T$

**begin**

**lemma** bound-inv-inv:

**assumes**

*inv S and*  
*n-d: no-dup (trail S) and*  
*atms-clss-S-A: atms-of-mm (clauses<sub>NOT</sub> S)  $\subseteq$  atms-of-ms A and*  
*atms-trail-S-A: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-ms A and*  
*finite A and*  
*cdcl<sub>NOT</sub>: cdcl<sub>NOT</sub> S T*  
**shows**  
*atms-of-mm (clauses<sub>NOT</sub> T)  $\subseteq$  atms-of-ms A and*  
*atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A and*  
*finite A*  
 <proof>

**sublocale** *cdcl<sub>NOT</sub>-increasing-restarts-ops*  $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) S \text{ cdcl}_{NOT} f$   
 $\lambda A S. \text{atms-of-mm (clauses}_{NOT} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l (trail S) } \subseteq \text{atms-of-ms } A \wedge$   
*finite A*  
 $\mu_{CDCL}' \lambda S. \text{inv } S \wedge \text{no-dup (trail S)}$   
 $\mu_{CDCL}'\text{-bound}$   
 <proof>

**lemma** *cdcl<sub>NOT</sub>-with-restart- $\mu_{CDCL}'$ -le- $\mu_{CDCL}'$ -bound:*

**assumes**

*cdcl<sub>NOT</sub>: cdcl<sub>NOT</sub>-restart (T, a) (V, b) and*  
*cdcl<sub>NOT</sub>-inv:*  
*inv T*  
*no-dup (trail T) and*  
*bound-inv:*  
*atms-of-mm (clauses<sub>NOT</sub> T)  $\subseteq$  atms-of-ms A*  
*atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A*  
*finite A*

**shows**  $\mu_{CDCL}' A V \leq \mu_{CDCL}'\text{-bound } A T$

<proof>

**lemma** *cdcl<sub>NOT</sub>-with-restart- $\mu_{CDCL}'$ -bound-le- $\mu_{CDCL}'$ -bound:*

**assumes**

*cdcl<sub>NOT</sub>: cdcl<sub>NOT</sub>-restart (T, a) (V, b) and*  
*cdcl<sub>NOT</sub>-inv:*  
*inv T*  
*no-dup (trail T) and*  
*bound-inv:*  
*atms-of-mm (clauses<sub>NOT</sub> T)  $\subseteq$  atms-of-ms A*  
*atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A*  
*finite A*

**shows**  $\mu_{CDCL}'\text{-bound } A V \leq \mu_{CDCL}'\text{-bound } A T$

<proof>

**sublocale** *cdcl<sub>NOT</sub>-increasing-restarts - - - -*

$f$   
 $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) S$   
 $\lambda A S. \text{atms-of-mm (clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{atm-of ' lits-of-l (trail S) } \subseteq \text{atms-of-ms } A \wedge \text{finite } A$   
 $\mu_{CDCL}' \text{ cdcl}_{NOT}$   
 $\lambda S. \text{inv } S \wedge \text{no-dup (trail S)}$   
 $\mu_{CDCL}'\text{-bound}$   
 <proof>

**lemma** *cdcl<sub>NOT</sub>-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 ( $trail$  ( $fst$   $T$ )))  
 $\langle proof \rangle$

**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 ( $trail$  ( $fst$   $T$ )))  
 $\langle proof \rangle$

**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**  $I \models_{sextm} clauses_{NOT} (fst S) \longleftrightarrow I \models_{sextm} clauses_{NOT} (fst T)$   
 $\langle proof \rangle$

**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**  $I \models_{sextm} clauses_{NOT} (fst S) \longleftrightarrow I \models_{sextm} clauses_{NOT} (fst T)$   
 $\langle proof \rangle$

**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- $mm$  ( $clauses_{NOT} S$ )  $\subseteq$   $atms$ -of- $ms$   $A$  **and**  
 $atms$ - $trail$ :  $atm$ -of ' $lits$ -of- $l$ ' ( $trail S$ )  $\subseteq$   $atms$ -of- $ms$   $A$  **and**  
 $n$ -d:  $no$ -dup ( $trail S$ ) **and**  
 $fin$ - $A[simp]$ :  $finite$   $A$  **and**  
 $inv$ :  $inv$   $S$  **and**  
 $decomp$ :  $all$ -decomposition-implies- $m$  ( $clauses_{NOT} S$ ) ( $get$ -all-ann-decomposition ( $trail S$ ))  
**shows**  $unsatisfiable$  ( $set$ - $mset$  ( $clauses_{NOT} S$ ))  
 $\vee$  ( $lits$ -of- $l$  ( $trail T$ )  $\models_{sextm} clauses_{NOT} S \wedge$   $satisfiable$  ( $set$ - $mset$  ( $clauses_{NOT} S$ )))  
 $\langle proof \rangle$   
**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  $trail$   $clauses_{NOT}$   $prepend$ - $trail$   $tl$ - $trail$   $add$ - $cls_{NOT}$   $remove$ - $cls_{NOT}$   
 $\lambda C$   $C'$   $L'$   $S$   $T$ .  $distinct$ - $mset$  ( $C' + \{\#L'\#\}$ )  $\wedge$   $backjump$ - $l$ - $cond$   $C$   $C'$   $L'$   $S$   $T$   
 $propagate$ - $conds$   $forget$ - $conds$   $inv$



**for**

$trail :: 'st \Rightarrow ('v, unit) \text{ ann-lits and}$   
 $clauses_{NOT} :: 'st \Rightarrow 'v \text{ clauses and}$   
 $prepend-trail :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $tl-trail :: 'st \Rightarrow 'st \text{ and}$   
 $add-cl_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $remove-cl_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$   
 $propagate-conds :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow bool \text{ and}$   
 $inv :: 'st \Rightarrow bool \text{ and}$   
 $forget-conds :: 'v \text{ clause} \Rightarrow 'st \Rightarrow bool \text{ and}$   
 $backjump-l-cond :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$   
 $+$

**fixes**  $f :: nat \Rightarrow nat$

**assumes**

$unbounded: unbounded\ f \text{ and } f\text{-ge-1}: \bigwedge n. n \geq 1 \Rightarrow f\ n \geq 1 \text{ and}$   
 $inv\text{-restart}: \bigwedge S\ T. inv\ S \Rightarrow T \sim reduce\text{-trail-to}_{NOT} \ \square\ S \Rightarrow inv\ T$

**begin**

**definition**  $not\text{-simplified-cl}\ A = \{\#C \in \# A. \text{tautology } C \vee \neg distinct\text{-mset } C\#\}$

**lemma**  $simple\text{-clss-or-not-simplified-cl}$ :

**assumes**  $atms\text{-of-mm}\ (clauses_{NOT}\ S) \subseteq atms\text{-of-ms}\ A$  **and**

$x \in \# clauses_{NOT}\ S$  **and**  $finite\ A$

**shows**  $x \in simple\text{-clss}\ (atms\text{-of-ms}\ A) \vee x \in \# not\text{-simplified-cl}\ (clauses_{NOT}\ S)$

$\langle proof \rangle$

**lemma**  $cdcl_{NOT}\text{-merged-bj-learn-clauses-bound}$ :

**assumes**

$cdcl_{NOT}\text{-merged-bj-learn}\ S\ T$  **and**

$inv: inv\ S$  **and**

$atms\text{-clss}: atms\text{-of-mm}\ (clauses_{NOT}\ S) \subseteq atms\text{-of-ms}\ A$  **and**

$atms\text{-trail}: atm\text{-of}\ ('(lits\text{-of-l}\ (trail\ S))) \subseteq atms\text{-of-ms}\ A$  **and**

$n\text{-d}: no\text{-dup}\ (trail\ S)$  **and**

$fin\text{-}A[simp]: finite\ A$

**shows**  $set\text{-mset}\ (clauses_{NOT}\ T) \subseteq set\text{-mset}\ (not\text{-simplified-cl}\ (clauses_{NOT}\ S))$

$\cup simple\text{-clss}\ (atms\text{-of-ms}\ A)$

$\langle proof \rangle$

**lemma**  $cdcl_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$ :

**assumes**  $cdcl_{NOT}\text{-merged-bj-learn}\ S\ T$

**shows**  $not\text{-simplified-cl}\ (clauses_{NOT}\ T) \subseteq \# not\text{-simplified-cl}\ (clauses_{NOT}\ S)$

$\langle proof \rangle$

**lemma**  $rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$ :

**assumes**  $cdcl_{NOT}\text{-merged-bj-learn}^{**}\ S\ T$

**shows**  $not\text{-simplified-cl}\ (clauses_{NOT}\ T) \subseteq \# not\text{-simplified-cl}\ (clauses_{NOT}\ S)$

$\langle proof \rangle$

**lemma**  $rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-clauses-bound}$ :

**assumes**

$cdcl_{NOT}\text{-merged-bj-learn}^{**}\ S\ T$  **and**

$inv\ S$  **and**

$atms\text{-of-mm}\ (clauses_{NOT}\ S) \subseteq atms\text{-of-ms}\ A$  **and**

$atm\text{-of}\ ('(lits\text{-of-l}\ (trail\ S))) \subseteq atms\text{-of-ms}\ A$  **and**

$n\text{-d}: no\text{-dup}\ (trail\ S)$  **and**

$finite[simp]: finite\ A$

**shows**  $\text{set-mset } (\text{clauses}_{NOT} T) \subseteq \text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S))$   
 $\cup \text{simple-clss } (\text{atms-of-ms } A)$   
 ⟨proof⟩

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

**lemma**  $\text{rtranchp-cdcl}_{NOT}\text{-merged-bj-learn-clauses-bound-card}$ :

**assumes**  
 $\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S \ T$  **and**  
 $\text{inv } S$  **and**  
 $\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$  **and**  
 $\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A$  **and**  
 $\text{no-dup } (\text{trail } S)$  **and**  
 $\text{finite: finite } A$   
**shows**  $\mu_{CDCL}'\text{-merged } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$   
 ⟨proof⟩

**sublocale**  $\text{cdcl}_{NOT}\text{-increasing-restarts-ops } \lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \ \text{list}) \ S$   
 $\text{cdcl}_{NOT}\text{-merged-bj-learn } f$   
 $\lambda A \ S. \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$   
 $\mu_{CDCL}'\text{-merged}$   
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$   
 $\mu_{CDCL}'\text{-bound}$   
 ⟨proof⟩

**lemma**  $\text{cdcl}_{NOT}\text{-restart-}\mu_{CDCL}'\text{-merged-le-}\mu_{CDCL}'\text{-bound}$ :

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

**lemma**  $\text{cdcl}_{NOT}\text{-restart-}\mu_{CDCL}'\text{-bound-le-}\mu_{CDCL}'\text{-bound}$ :

**assumes**  
 $\text{cdcl}_{NOT}\text{-restart } T \ V$  **and**  
 $\text{no-dup } (\text{trail } (\text{fst } T))$  **and**  
 $\text{inv } (\text{fst } T)$  **and**  
 $\text{fin: finite } A$   
**shows**  $\mu_{CDCL}'\text{-bound } A \ (\text{fst } V) \leq \mu_{CDCL}'\text{-bound } A \ (\text{fst } T)$   
 ⟨proof⟩

**sublocale**  $\text{cdcl}_{NOT}\text{-increasing-restarts - - - - } f$   
 $\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \ \text{list}) \ S$   
 $\lambda A \ S. \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$   
 $\wedge \text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$   
 $\mu_{CDCL}'\text{-merged } \text{cdcl}_{NOT}\text{-merged-bj-learn}$   
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$

$\lambda A \ T. ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$   
 $+ \text{card } (\text{set-mset } (\text{not-simplified-clauses}_{NOT} \ T))$   
 $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>NOT</sub>-restart-eq-sat-iff*:

**assumes**

*cdcl<sub>NOT</sub>-restart*  $S \ T$  **and**  
*no-dup* (*trail* (*fst*  $S$ ))  
*inv* (*fst*  $S$ )

**shows**  $I \models_{\text{sextm}} \text{clauses}_{NOT} \ (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} \ (\text{fst } T)$

$\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-restart-eq-sat-iff*:

**assumes**

*cdcl<sub>NOT</sub>-restart\*\**  $S \ T$  **and**  
*inv*: *inv* (*fst*  $S$ ) **and** *n-d*: *no-dup*(*trail* (*fst*  $S$ ))

**shows**  $I \models_{\text{sextm}} \text{clauses}_{NOT} \ (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} \ (\text{fst } T)$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>NOT</sub>-restart-all-decomposition-implies-m*:

**assumes**

*cdcl<sub>NOT</sub>-restart*  $S \ T$  **and**  
*inv*: *inv* (*fst*  $S$ ) **and** *n-d*: *no-dup*(*trail* (*fst*  $S$ )) **and**  
*all-decomposition-implies-m* (*clauses<sub>NOT</sub>* (*fst*  $S$ ))  
*(get-all-ann-decomposition* (*trail* (*fst*  $S$ )))

**shows** *all-decomposition-implies-m* (*clauses<sub>NOT</sub>* (*fst*  $T$ ))

*(get-all-ann-decomposition* (*trail* (*fst*  $T$ )))

$\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-restart-all-decomposition-implies-m*:

**assumes**

*cdcl<sub>NOT</sub>-restart\*\**  $S \ T$  **and**  
*inv*: *inv* (*fst*  $S$ ) **and** *n-d*: *no-dup*(*trail* (*fst*  $S$ )) **and**  
*decomp*: *all-decomposition-implies-m* (*clauses<sub>NOT</sub>* (*fst*  $S$ ))  
*(get-all-ann-decomposition* (*trail* (*fst*  $S$ )))

**shows** *all-decomposition-implies-m* (*clauses<sub>NOT</sub>* (*fst*  $T$ ))

*(get-all-ann-decomposition* (*trail* (*fst*  $T$ )))

$\langle \text{proof} \rangle$

**lemma** *full-cdcl<sub>NOT</sub>-restart-normal-form*:

**assumes**

*full*: *full cdcl<sub>NOT</sub>-restart*  $S \ T$  **and**  
*inv*: *inv* (*fst*  $S$ ) **and** *n-d*: *no-dup*(*trail* (*fst*  $S$ )) **and**  
*decomp*: *all-decomposition-implies-m* (*clauses<sub>NOT</sub>* (*fst*  $S$ ))  
*(get-all-ann-decomposition* (*trail* (*fst*  $S$ ))) **and**  
*atms-cls*: *atms-of-mm* (*clauses<sub>NOT</sub>* (*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<sub>NOT</sub>* (*fst*  $S$ )))

$\vee$  *lits-of-l* (*trail* (*fst*  $T$ ))  $\models_{\text{sextm}} \text{clauses}_{NOT} \ (\text{fst } S) \wedge$

*satisfiable* (*set-mset* (*clauses<sub>NOT</sub>* (*fst*  $S$ )))

$\langle \text{proof} \rangle$

**corollary** *full-cdcl<sub>NOT</sub>-restart-normal-form-init-state*:

**assumes**

```

  init-state: trail S = [] clausesNOT S = N and
  full: full cdclNOT-restart (S, 0) T and
  inv: inv S
shows unsatisfiable (set-mset N)
  ∨ lits-of-l (trail (fst T))  $\models_{sextm}$  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  $\models_{as}$  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  $\models_{as}$  CNot C ∧

```

```

    undefined-lit F L ∧ atm-of L ∈ atms-of-mm N ∪ atm-of ' lits-of-l (F' @ Decided K # F) ∧

```

```

    N  $\models_{pm}$  C' + {#L#} ∧ F  $\models_{as}$  CNot C'

```

```

⟨proof⟩

```

```

lemma backtrack-is-backjump':

```

```

  fixes M M' :: ('v, unit) ann-lits

```

```

  assumes

```

```

    backtrack: backtrack S T and

```

```

    no-dup: (no-dup ∘ 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  $\models_{as}$  CNot C

```

```

    ∧ undefined-lit F L ∧ atm-of L ∈ atms-of-mm (snd S) ∪ atm-of ' lits-of-l (fst S) ∧

```

```

    snd S  $\models_{pm}$  C' + {#L#} ∧ F  $\models_{as}$  CNot C'

```

```

⟨proof⟩

```

**sublocale** *dpll-state*

*fst snd*  $\lambda L (M, N). (L \# M, N) \lambda(M, N). (tl M, N)$   
 $\lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset C N)$   
 $\langle proof \rangle$

**sublocale** *backjumping-ops*

*fst snd*  $\lambda L (M, N). (L \# M, N) \lambda(M, N). (tl M, N)$   
 $\lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset C N) \lambda- - S T. backtrack S T$   
 $\langle proof \rangle$

**thm** *reduce-trail-to<sub>NOT</sub>-clauses*

**lemma** *reduce-trail-to<sub>NOT</sub>:*

*reduce-trail-to<sub>NOT</sub>*  $F S =$   
*(if length (fst S)  $\geq$  length F*  
*then drop (length (fst S) - length F) (fst S)*  
*else []*,  
*snd S) (is ?R = ?C)*

$\langle proof \rangle$

**lemma** *backtrack-is-backjump'':*

**fixes**  $M M' :: ('v, unit) ann-lits$

**assumes**

*backtrack:* *backtrack S T and*

*no-dup:* *(no-dup  $\circ$  fst) S and*

*decomp:* *all-decomposition-implies-m (snd S) (get-all-ann-decomposition (fst S))*

**shows** *backjump S T*

$\langle proof \rangle$

**lemma** *can-do-bt-step:*

**assumes**

$M: fst S = F' @ Decided K \# F$  **and**

$C \in \# snd S$  **and**

$C: fst S \models_{as} CNot C$

**shows**  $\neg no-step backtrack S$

$\langle proof \rangle$

**end**

**sublocale** *dpll-with-backtrack*  $\subseteq$  *dpll-with-backjumping-ops*

*fst snd*  $\lambda L (M, N). (L \# M, N)$   
 $\lambda(M, N). (tl M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset C N)$   
 $\lambda(M, N). no-dup M \wedge all-decomposition-implies-m N (get-all-ann-decomposition M)$   
 $\lambda- - S T. backtrack S T$   
 $\lambda- -. True$   
 $\langle proof \rangle$

**sublocale** *dpll-with-backtrack*  $\subseteq$  *dpll-with-backjumping*

*fst snd*  $\lambda L (M, N). (L \# M, N)$   
 $\lambda(M, N). (tl M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset C N)$   
 $\lambda(M, N). no-dup M \wedge all-decomposition-implies-m N (get-all-ann-decomposition M)$   
 $\lambda- - S T. backtrack S T$   
 $\lambda- -. True$   
 $\langle proof \rangle$

**context** *dpll-with-backtrack*

**begin**

**lemma** *wf-tranclp-dpll-initail-state:*

**assumes** *fin: finite A*

**shows** *wf {((M':('v, unit) ann-lits, N':('v clauses), ([], N))|M' N' N.*

*dpll-bj<sup>++</sup> ([], 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' ⊨<sub>asm</sub> 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' ⊨<sub>asm</sub> 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<sub>NOT</sub>-is-dpll:*

*cdcl<sub>NOT</sub> S T ⟷ dpll-bj S T*

*⟨proof⟩*

Another proof of termination:

**lemma** *wf {(T, S). dpll-bj S T ∧ cdcl<sub>NOT</sub>-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 ≥ 1*

**begin**

**sublocale** *cdcl<sub>NOT</sub>-increasing-restarts*

*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) f λ(-, N) S. S = ([], N)
  λA (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)
  λA T. (2+card (atms-of-ms A)) ^ (1+card (atms-of-ms A))
    - μC (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)
  λA -. (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 dpllW-ann-lit = ('a, unit) ann-lit
type-synonym 'a dpllW-ann-lits = ('a, unit) ann-lits
type-synonym 'v dpllW-state = 'v dpllW-ann-lits × 'v clauses

```

**abbreviation** *trail* :: 'v dpll<sub>W</sub>-state ⇒ 'v dpll<sub>W</sub>-ann-lits **where**  
*trail* ≡ *fst*

**abbreviation** *clauses* :: 'v dpll<sub>W</sub>-state ⇒ 'v clauses **where**  
*clauses* ≡ *snd*

**inductive** *dpll<sub>W</sub>* :: 'v dpll<sub>W</sub>-state ⇒ 'v dpll<sub>W</sub>-state ⇒ bool **where**  
*propagate*: *C* + {#*L*#} ∈ # *clauses* *S* ⇒ *trail* *S* ⊨<sub>as</sub> *CNot* *C* ⇒ undefined-lit (*trail* *S*) *L*  
 ⇒ *dpll<sub>W</sub>* *S* (*Propagated* *L* () # *trail* *S*, *clauses* *S*) |  
*decided*: undefined-lit (*trail* *S*) *L* ⇒ atm-of *L* ∈ atms-of-mm (*clauses* *S*)  
 ⇒ *dpll<sub>W</sub>* *S* (*Decided* *L* # *trail* *S*, *clauses* *S*) |  
*backtrack*: *backtrack-split* (*trail* *S*) = (*M'*, *L* # *M*) ⇒ is-decided *L* ⇒ *D* ∈ # *clauses* *S*  
 ⇒ *trail* *S* ⊨<sub>as</sub> *CNot* *D* ⇒ *dpll<sub>W</sub>* *S* (*Propagated* (- (lit-of *L*)) () # *M*, *clauses* *S*)

### 5.4.2 Invariants

**lemma** *dpll<sub>W</sub>-distinct-inv*:  
 assumes *dpll<sub>W</sub>* *S* *S'*  
 and no-dup (*trail* *S*)  
 shows no-dup (*trail* *S'*)  
 ⟨proof⟩

**lemma** *dpll<sub>W</sub>-consistent-interp-inv*:  
 assumes *dpll<sub>W</sub>* *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<sub>W</sub>-vars-in-snd-inv*:  
 assumes *dpll<sub>W</sub>* *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'*)

$\langle \text{proof} \rangle$

**lemma** *atms-of-ms-lit-of-atms-of*:  $\text{atms-of-ms } ((\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' } c) = \text{atm-of ' lit-of ' } c$   
 $\langle \text{proof} \rangle$

theorem 2.8.2 page 73 of Weidenbach's book

**lemma** *dpll<sub>W</sub>-propagate-is-conclusion*:  
  **assumes** *dpll<sub>W</sub> S S'*  
  **and** *all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S))*  
  **and** *atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-mm (clauses S)*  
  **shows** *all-decomposition-implies-m (clauses S') (get-all-ann-decomposition (trail S'))*  
 $\langle \text{proof} \rangle$

theorem 2.8.3 page 73 of Weidenbach's book

**theorem** *dpll<sub>W</sub>-propagate-is-conclusion-of-decided*:  
  **assumes** *dpll<sub>W</sub> S S'*  
  **and** *all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S))*  
  **and** *atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-mm (clauses S)*  
  **shows**  $\text{set-mset (clauses S')} \cup \{\{\# \text{lit-of } L \# \} \mid L. \text{ is-decided } L \wedge L \in \text{set (trail S')}\}$   
     $\models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' } \bigcup (\text{set ' snd ' set (get-all-ann-decomposition (trail S'))})$   
 $\langle \text{proof} \rangle$

theorem 2.8.4 page 73 of Weidenbach's book

**lemma** *only-propagated-vars-unsat*:  
  **assumes** *decided:  $\forall x \in \text{set } M. \neg \text{is-decided } x$*   
  **and** *DN:  $D \in N$  and  $D: M \models_{as} C \text{Not } D$*   
  **and** *inv: all-decomposition-implies N (get-all-ann-decomposition M)*  
  **and** *atm-incl: atm-of ' lits-of-l M  $\subseteq$  atms-of-ms N*  
  **shows** *unsatisfiable N*  
 $\langle \text{proof} \rangle$

**lemma** *dpll<sub>W</sub>-same-clauses*:  
  **assumes** *dpll<sub>W</sub> S S'*  
  **shows** *clauses S = clauses S'*  
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-dpll<sub>W</sub>-inv*:  
  **assumes** *rtranclp dpll<sub>W</sub> 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)  $\subseteq$  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')  $\subseteq$  atms-of-mm (clauses S')*  
  **and** *clauses S = clauses S'*  
  **and** *consistent-interp (lits-of-l (trail S'))*  
  **and** *no-dup (trail S')*  
 $\langle \text{proof} \rangle$

**definition** *dpll<sub>W</sub>-all-inv S  $\equiv$*   
  *(all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S)))*  
   $\wedge$  *atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-mm (clauses S)*  
   $\wedge$  *consistent-interp (lits-of-l (trail S))*  
   $\wedge$  *no-dup (trail S)*



**lemma** *dpll<sub>W</sub>-all-inv-dest*[*dest*]:  
**assumes** *dpll<sub>W</sub>-all-inv* *S*  
**shows** *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))  
**and** *atm-of* ‘ *lits-of-l* (*trail S*)  $\subseteq$  *atms-of-mm* (*clauses S*)

**and** *consistent-interp* (*lits-of-l* (*trail S*))  $\wedge$  *no-dup* (*trail S*)  
 $\langle$ *proof* $\rangle$

**lemma** *rtrancpl-dpll<sub>W</sub>-all-inv*:  
**assumes** *rtrancpl dpll<sub>W</sub> S S'*  
**and** *dpll<sub>W</sub>-all-inv S*  
**shows** *dpll<sub>W</sub>-all-inv S'*  
 $\langle$ *proof* $\rangle$

**lemma** *dpll<sub>W</sub>-all-inv*:  
**assumes** *dpll<sub>W</sub> S S'*  
**and** *dpll<sub>W</sub>-all-inv S*  
**shows** *dpll<sub>W</sub>-all-inv S'*  
 $\langle$ *proof* $\rangle$

**lemma** *rtrancpl-dpll<sub>W</sub>-inv-starting-from-0*:  
**assumes** *rtrancpl dpll<sub>W</sub> S S'*  
**and** *inv: trail S = []*  
**shows** *dpll<sub>W</sub>-all-inv S'*  
 $\langle$ *proof* $\rangle$

**lemma** *dpll<sub>W</sub>-can-do-step*:  
**assumes** *consistent-interp* (*set M*)  
**and** *distinct M*  
**and** *atm-of* ‘ (*set M*)  $\subseteq$  *atms-of-mm N*  
**shows** *rtrancpl dpll<sub>W</sub> ([], N)* (*map Decided M, N*)  
 $\langle$ *proof* $\rangle$

**definition** *conclusive-dpll<sub>W</sub>-state* (*S:: 'v dpll<sub>W</sub>-state*)  $\longleftrightarrow$   
(*trail S*  $\models_{asm}$  *clauses S*  $\vee$  ( $\forall L \in \text{set } (\text{trail } S). \neg is-decided L$ )  
 $\wedge$  ( $\exists C \in \# \text{ clauses } S. \text{trail } S \models_{as} CNot C$ )))

theorem 2.8.6 page 74 of Weidenbach’s book

**lemma** *dpll<sub>W</sub>-strong-completeness*:  
**assumes** *set M*  $\models_{sm} N$   
**and** *consistent-interp* (*set M*)  
**and** *distinct M*  
**and** *atm-of* ‘ (*set M*)  $\subseteq$  *atms-of-mm N*  
**shows** *dpll<sub>W</sub>\*\** ([], *N*) (*map Decided M, N*)  
**and** *conclusive-dpll<sub>W</sub>-state* (*map Decided M, N*)  
 $\langle$ *proof* $\rangle$

theorem 2.8.5 page 73 of Weidenbach’s book

**lemma** *dpll<sub>W</sub>-sound*:  
**assumes**  
*rtrancpl dpll<sub>W</sub> ([], N)* (*M, N*) **and**  
 $\forall S. \neg dpll_W (M, N) S$   
**shows** *M*  $\models_{asm} N \longleftrightarrow$  *satisfiable* (*set-mset N*) (**is** ?*A*  $\longleftrightarrow$  ?*B*)  
 $\langle$ *proof* $\rangle$

### 5.4.3 Termination

**definition**  $dpll_W\text{-mes } M \ n =$

$\text{map } (\lambda l. \text{ if is-decided } l \text{ then } 2 \text{ else } (1::\text{nat})) (\text{rev } M) \ @ \ \text{replicate } (n - \text{length } M) \ 3$

**lemma**  $\text{length-dpll}_W\text{-mes}$ :

**assumes**  $\text{length } M \leq n$

**shows**  $\text{length } (dpll_W\text{-mes } M \ n) = n$

$\langle \text{proof} \rangle$

**lemma**  $\text{distinctcard-atm-of-lit-of-eq-length}$ :

**assumes**  $\text{no-dup } S$

**shows**  $\text{card } (\text{atm-of } \text{' lits-of-l } S) = \text{length } S$

$\langle \text{proof} \rangle$

**lemma**  $dpll_W\text{-card-decrease}$ :

**assumes**  $dpll$ :  $dpll_W \ S \ S'$  **and**  $\text{length } (\text{trail } S') \leq \text{card vars}$

**and**  $\text{length } (\text{trail } S) \leq \text{card vars}$

**shows**  $(dpll_W\text{-mes } (\text{trail } S') \ (\text{card vars}), dpll_W\text{-mes } (\text{trail } S) \ (\text{card vars}))$

$\in \text{lexn } \{(a, b). a < b\} \ (\text{card vars})$

$\langle \text{proof} \rangle$

theorem 2.8.7 page 74 of Weidenbach's book

**lemma**  $dpll_W\text{-card-decrease'}$ :

**assumes**  $dpll$ :  $dpll_W \ S \ S'$

**and**  $\text{atm-incl}$ :  $\text{atm-of } \text{' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-mm } (\text{clauses } S)$

**and**  $\text{no-dup}$ :  $\text{no-dup } (\text{trail } S)$

**shows**  $(dpll_W\text{-mes } (\text{trail } S') \ (\text{card } (\text{atms-of-mm } (\text{clauses } S'))),$

$dpll_W\text{-mes } (\text{trail } S) \ (\text{card } (\text{atms-of-mm } (\text{clauses } S)))) \in \text{lex } \{(a, b). a < b\}$

$\langle \text{proof} \rangle$

**lemma**  $\text{wf-lexn}$ :  $\text{wf } (\text{lexn } \{(a, b). (a::\text{nat}) < b\} \ (\text{card } (\text{atms-of-mm } (\text{clauses } S))))$

$\langle \text{proof} \rangle$

**lemma**  $dpll_W\text{-wf}$ :

$\text{wf } \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}$

$\langle \text{proof} \rangle$

**lemma**  $dpll_W\text{-tranclp-star-commute}$ :

$\{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}^+ = \{(S', S). dpll_W\text{-all-inv } S \wedge \text{tranclp } dpll_W \ S \ S'\}$

(**is**  $?A = ?B$ )

$\langle \text{proof} \rangle$

**lemma**  $dpll_W\text{-wf-tranclp}$ :  $\text{wf } \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W^{++} \ S \ S'\}$

$\langle \text{proof} \rangle$

**lemma**  $dpll_W\text{-wf-plus}$ :

**shows**  $\text{wf } \{(S', ([], N)) | S'. dpll_W^{++} \ ([], N) \ S'\} \ (\text{is } \text{wf } ?P)$

$\langle \text{proof} \rangle$

### 5.4.4 Final States

Proposition 2.8.1: final states are the normal forms of  $dpll_W$

**lemma**  $dpll_W\text{-no-more-step-is-a-conclusive-state}$ :

**assumes**  $\forall S'. \neg dpll_W S S'$   
**shows** *conclusive-dpll<sub>W</sub>-state*  $S$   
 $\langle proof \rangle$

**lemma** *dpll<sub>W</sub>-conclusive-state-correct*:  
**assumes**  $dpll_W^{**} ([], N) (M, N)$  **and** *conclusive-dpll<sub>W</sub>-state*  $(M, N)$   
**shows**  $M \models_{asm} N \longleftrightarrow \text{satisfiable } (set\text{-}mset\ N)$  (**is**  $?A \longleftrightarrow ?B$ )  
 $\langle proof \rangle$

### 5.4.5 Link with NOT's DPLL

**interpretation** *dpll<sub>W</sub>-NOT*: *dpll-with-backtrack*  $\langle proof \rangle$

**declare** *dpll<sub>W</sub>-NOT.state-simp<sub>NOT</sub>*[*simp del*]  
**lemma** *state-eq<sub>NOT</sub>-iff-eq*[*iff, simp*]: *dpll<sub>W</sub>-NOT.state-eq<sub>NOT</sub>*  $S\ T \longleftrightarrow S = T$   
 $\langle proof \rangle$

**lemma** *dpll<sub>W</sub>-dpll<sub>W</sub>-bj*:  
**assumes** *inv*: *dpll<sub>W</sub>-all-inv*  $S$  **and** *dpll*: *dpll<sub>W</sub>*  $S\ T$   
**shows** *dpll<sub>W</sub>-NOT.dpll-bj*  $S\ T$   
 $\langle proof \rangle$

**lemma** *dpll<sub>W</sub>-bj-dpll*:  
**assumes** *inv*: *dpll<sub>W</sub>-all-inv*  $S$  **and** *dpll*: *dpll<sub>W</sub>-NOT.dpll-bj*  $S\ T$   
**shows** *dpll<sub>W</sub>*  $S\ T$   
 $\langle proof \rangle$

**lemma** *rtrancp-dpll<sub>W</sub>-rtrancp-dpll<sub>W</sub>-NOT*:  
**assumes**  $dpll_W^{**} S\ T$  **and** *dpll<sub>W</sub>-all-inv*  $S$   
**shows** *dpll<sub>W</sub>-NOT.dpll-bj<sup>\*\*</sup>*  $S\ T$   
 $\langle proof \rangle$

**lemma** *rtrancp-dpll-rtrancp-dpll<sub>W</sub>*:  
**assumes** *dpll<sub>W</sub>-NOT.dpll-bj<sup>\*\*</sup>*  $S\ T$  **and** *dpll<sub>W</sub>-all-inv*  $S$   
**shows**  $dpll_W^{**} S\ T$   
 $\langle proof \rangle$

**lemma** *dpll-conclusive-state-correctness*:  
**assumes**  $dpll_W\text{-}NOT.dpll\text{-}bj^{**} ([], N) (M, N)$  **and** *conclusive-dpll<sub>W</sub>-state*  $(M, N)$   
**shows**  $M \models_{asm} N \longleftrightarrow \text{satisfiable } (set\text{-}mset\ N)$   
 $\langle proof \rangle$

**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* ::  $('v, 'm) \text{ ann-lits} \Rightarrow \text{nat}$  **where**  
*count-decided*  $l \equiv \text{length } (\text{filter is-decided } l)$

**abbreviation** *get-level* ::  $('v, 'm) \text{ ann-lits} \Rightarrow 'v \text{ literal} \Rightarrow \text{nat}$  **where**  
*get-level*  $S\ L \equiv \text{length } (\text{filter is-decided } (\text{dropWhile } (\lambda S. \text{atm-of } (\text{lit-of } S) \neq \text{atm-of } L) S))$

**lemma** *get-level-uminus*:  $\text{get-level } M \ (-L) = \text{get-level } M \ L$   
 $\langle \text{proof} \rangle$

**lemma** *atm-of-notin-get-rev-level-eq-0[simp]*:  
**assumes**  $\text{atm-of } L \notin \text{atm-of ' lits-of-l } M$   
**shows**  $\text{get-level } M \ L = 0$   
 $\langle \text{proof} \rangle$

**lemma** *get-level-ge-0-atm-of-in*:  
**assumes**  $\text{get-level } M \ L > n$   
**shows**  $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$   
 $\langle \text{proof} \rangle$

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**  $\text{atm-of } L \notin \text{atm-of ' lits-of-l } M$   
**shows**  $\text{get-level } (M \ @ \ M') \ L = \text{get-level } M' \ L$   
 $\langle \text{proof} \rangle$

If the literal is at the beginning, then the end can be skipped

**lemma** *get-rev-level-skip-end[simp]*:  
**assumes**  $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$   
**shows**  $\text{get-level } (M \ @ \ M') \ L = \text{get-level } M \ L + \text{length } (\text{filter is-decided } M')$   
 $\langle \text{proof} \rangle$

**lemma** *get-level-skip-beginning*:  
**assumes**  $\text{atm-of } L' \neq \text{atm-of } (\text{lit-of } K)$   
**shows**  $\text{get-level } (K \ # \ M) \ L' = \text{get-level } M \ L'$   
 $\langle \text{proof} \rangle$

**lemma** *get-level-skip-beginning-not-decided[simp]*:  
**assumes**  $\text{atm-of } L \notin \text{atm-of ' lits-of-l } S$   
**and**  $\forall s \in \text{set } S. \neg \text{is-decided } s$   
**shows**  $\text{get-level } (M \ @ \ S) \ L = \text{get-level } M \ L$   
 $\langle \text{proof} \rangle$

**lemma** *get-level-skip-in-all-not-decided*:  
**fixes**  $M :: ('a, 'b) \text{ ann-lits}$  **and**  $L :: 'a \text{ literal}$   
**assumes**  $\forall m \in \text{set } M. \neg \text{is-decided } m$   
**and**  $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$   
**shows**  $\text{get-level } M \ L = 0$   
 $\langle \text{proof} \rangle$

**lemma** *get-level-skip-all-not-decided[simp]*:  
**fixes**  $M$   
**assumes**  $\forall m \in \text{set } M. \neg \text{is-decided } m$   
**shows**  $\text{get-level } M \ L = 0$   
 $\langle \text{proof} \rangle$

**abbreviation**  $M\text{Max } M \equiv \text{Max } (\text{set-mset } M)$

the  $\{\#0::'a\# \}$  is there to ensure that the set is not empty.

**definition** *get-maximum-level* ::  $('a, 'b) \text{ ann-lits} \Rightarrow 'a \text{ literal multiset} \Rightarrow \text{nat}$

**where**  
 $get\_maximum\_level\ M\ D = MMax\ (\{\#0\# \} + image\_mset\ (get\_level\ M)\ D)$

**lemma** *get-maximum-level-ge-get-level*:  
 $L \in \# D \implies get\_maximum\_level\ M\ D \geq get\_level\ M\ L$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-empty[simp]*:  
 $get\_maximum\_level\ M\ \{\#\} = 0$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-exists-lit-of-max-level*:  
 $D \neq \{\#\} \implies \exists L \in \# D. get\_level\ M\ L = get\_maximum\_level\ M\ D$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-empty-list[simp]*:  
 $get\_maximum\_level\ []\ D = 0$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-single[simp]*:  
 $get\_maximum\_level\ M\ \{\#L\# \} = get\_level\ M\ L$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-plus*:  
 $get\_maximum\_level\ M\ (D + D') = \max\ (get\_maximum\_level\ M\ D)\ (get\_maximum\_level\ M\ D')$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-exists-lit*:  
**assumes**  $n: n > 0$   
**and**  $max: get\_maximum\_level\ M\ D = n$   
**shows**  $\exists L \in \# D. get\_level\ M\ L = n$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-skip-first[simp]*:  
**assumes**  $atm\_of\ L \notin atms\_of\ D$   
**shows**  $get\_maximum\_level\ (Propagated\ L\ C\ \# M)\ D = get\_maximum\_level\ M\ D$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-skip-beginning*:  
**assumes**  $DH: \forall x \in atms\_of\ D. x \notin atm\_of\ ' lits\_of\_l\ c$   
**shows**  $get\_maximum\_level\ (c\ @\ H)\ D = get\_maximum\_level\ H\ D$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-D-single-propagated*:  
 $get\_maximum\_level\ [Propagated\ x21\ x22]\ D = 0$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-skip-un-decided-not-present*:  
**assumes**  
 $\forall L \in \# D. atm\_of\ L \notin atm\_of\ ' lits\_of\_l\ M$  **and**  
 $\forall m \in set\ M. \neg is\_decided\ m$   
**shows**  $get\_maximum\_level\ (M\ @\ aa)\ D = get\_maximum\_level\ aa\ D$   
 $\langle proof \rangle$

**lemma** *get-maximum-level-union-mset*:  
 $get\_maximum\_level\ M\ (A\ \#\cup\ B) = get\_maximum\_level\ M\ (A + B)$

$\langle \text{proof} \rangle$

**lemma** *count-decided-rev[simp]*:  
  *count-decided* (rev *M*) = *count-decided* *M*  
 $\langle \text{proof} \rangle$

**lemma** *count-decided-ge-get-level[simp]*:  
  *count-decided* *M*  $\geq$  *get-level* *M* *L*  
 $\langle \text{proof} \rangle$

**lemma** *count-decided-ge-get-maximum-level*:  
  *count-decided* *M*  $\geq$  *get-maximum-level* *M* *D*  
 $\langle \text{proof} \rangle$

**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*  
 $\langle \text{proof} \rangle$

## Properties about the levels

**lemma** *atm-lit-of-set-lits-of-l*:  
  ( $\lambda l. \text{atm-of } (\text{lit-of } l)$ ) ‘ *set* *xs* = *atm-of* ‘ *lits-of-l* *xs*  
 $\langle \text{proof} \rangle$

**lemma** *le-count-decided-decomp*:  
  **assumes** *no-dup* *M*  
  **shows**  $i < \text{count-decided } M \longleftrightarrow (\exists c \ K \ c'. M = c @ \text{Decided } K \ \# \ c' \wedge \text{get-level } M \ K = \text{Suc } i)$   
  **(is** ?*A*  $\longleftrightarrow$  ?*B*)  
 $\langle \text{proof} \rangle$

**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( $\mathbb{Z}$ )[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 stateW-ops =
```

**fixes**

*trail* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits **and**  
*init-clss* :: 'st  $\Rightarrow$  'v clauses **and**  
*learned-clss* :: 'st  $\Rightarrow$  'v clauses **and**  
*backtrack-lvl* :: 'st  $\Rightarrow$  nat **and**  
*conflicting* :: 'st  $\Rightarrow$  'v clause option **and**

*cons-trail* :: ('v, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-learned-clss* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-clss* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*update-backtrack-lvl* :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*update-conflicting* :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*init-state* :: 'v clauses  $\Rightarrow$  'st

**begin**

**abbreviation** *hd-trail* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lit **where**  
*hd-trail* *S*  $\equiv$  *hd* (*trail* *S*)

**definition** *clauses* :: 'st  $\Rightarrow$  'v clauses **where**  
*clauses* *S* = *init-clss* *S* + *learned-clss* *S*

**abbreviation** *resolve-clss* **where**

*resolve-clss* *L* *D'* *E*  $\equiv$  *remove1-mset* ( $-L$ ) *D'*  $\# \cup$  *remove1-mset* *L* *E*

**abbreviation** *state* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits  $\times$  'v clauses  $\times$  'v clauses  
 $\times$  nat  $\times$  'v clause option **where**

*state* *S*  $\equiv$  (*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<sub>W</sub>* =



*stateW-ops*

— functions about the state:

— getter:

*trail init-clss learned-clss backtrack-lvl conflicting*

— setter:

*cons-trail tl-trail add-learned-clss remove-clss update-backtrack-lvl  
update-conflicting*

— Some specific states:

*init-state*

**for**

*trail* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits **and**

*init-clss* :: 'st  $\Rightarrow$  'v clauses **and**

*learned-clss* :: 'st  $\Rightarrow$  'v clauses **and**

*backtrack-lvl* :: 'st  $\Rightarrow$  nat **and**

*conflicting* :: 'st  $\Rightarrow$  'v clause option **and**

*cons-trail* :: ('v, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*tl-trail* :: 'st  $\Rightarrow$  'st **and**

*add-learned-clss* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*remove-clss* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*update-backtrack-lvl* :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*update-conflicting* :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*init-state* :: 'v clauses  $\Rightarrow$  'st +

**assumes**

*cons-trail*:

$\bigwedge S'. \text{state } st = (M, S') \implies$   
state (cons-trail L st) = (L # M, S') **and**

*tl-trail*:

$\bigwedge S'. \text{state } st = (M, S') \implies \text{state } (tl\text{-trail } st) = (tl\ M, S') \text{ and}$

*remove-clss*:

$\bigwedge S'. \text{state } st = (M, N, U, S') \implies$   
state (remove-clss C st) =  
(M, removeAll-mset C N, removeAll-mset C U, S') **and**

*add-learned-clss*:

$\bigwedge S'. \text{state } st = (M, N, U, S') \implies$   
state (add-learned-clss C st) = (M, N, {#C#} + U, S') **and**

*update-backtrack-lvl*:

$\bigwedge S'. \text{state } st = (M, N, U, k, S') \implies$   
state (update-backtrack-lvl k' st) = (M, N, U, k', S') **and**

*update-conflicting*:

state st = (M, N, U, k, D)  $\implies$   
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]:  
 $\text{conflicting } (\text{remove-cls } C \text{ st}) = \text{conflicting st}$  **and**  
*conflicting-update-backtrack-lvl*[simp]:  
 $\text{conflicting } (\text{update-backtrack-lvl } k \text{ st}) = \text{conflicting st}$  **and**  
*conflicting-update-conflicting*[simp]:  
 $\text{conflicting } (\text{update-conflicting } E \text{ st}) = E$  **and**  
*init-state-trail*[simp]:  $\text{trail } (\text{init-state } N) = []$  **and**  
*init-state-clss*[simp]:  $\text{init-clss } (\text{init-state } N) = N$  **and**  
*init-state-learned-clss*[simp]:  $\text{learned-clss } (\text{init-state } N) = \{\#\}$  **and**  
*init-state-backtrack-lvl*[simp]:  $\text{backtrack-lvl } (\text{init-state } N) = 0$  **and**  
*init-state-conflicting*[simp]:  $\text{conflicting } (\text{init-state } N) = \text{None}$

$\langle \text{proof} \rangle$

**lemma**

**shows**

*clauses-cons-trail*[simp]:  
 $\text{clauses } (\text{cons-trail } M \text{ S}) = \text{clauses S}$  **and**  
*clss-tl-trail*[simp]:  $\text{clauses } (\text{tl-trail } S) = \text{clauses S}$  **and**  
*clauses-add-learned-cls-unfolded*:  
 $\text{clauses } (\text{add-learned-cls } U \text{ S}) = \{\#U\# \} + \text{learned-clss S} + \text{init-clss S}$   
**and**  
*clauses-update-backtrack-lvl*[simp]:  $\text{clauses } (\text{update-backtrack-lvl } k \text{ S}) = \text{clauses S}$  **and**  
*clauses-update-conflicting*[simp]:  $\text{clauses } (\text{update-conflicting } D \text{ S}) = \text{clauses S}$  **and**  
*clauses-remove-cls*[simp]:  
 $\text{clauses } (\text{remove-cls } C \text{ S}) = \text{removeAll-mset } C \text{ (clauses S)}$  **and**  
*clauses-add-learned-cls*[simp]:  
 $\text{clauses } (\text{add-learned-cls } C \text{ S}) = \{\#C\# \} + \text{clauses S}$  **and**  
*clauses-init-state*[simp]:  $\text{clauses } (\text{init-state } N) = N$   
 $\langle \text{proof} \rangle$

**abbreviation**  $\text{incr-lvl} :: 'st \Rightarrow 'st$  **where**

$\text{incr-lvl } S \equiv \text{update-backtrack-lvl } (\text{backtrack-lvl } S + 1) \text{ S}$

**definition**  $\text{state-eq} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  (**infix**  $\sim$  50) **where**

$S \sim T \longleftrightarrow \text{state } S = \text{state } T$

**lemma** *state-eq-ref*[simp, intro]:

$S \sim S$

$\langle \text{proof} \rangle$

**lemma** *state-eq-sym*:

$S \sim T \longleftrightarrow T \sim S$

$\langle \text{proof} \rangle$

**lemma** *state-eq-trans*:

$S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$

$\langle \text{proof} \rangle$

**lemma**

**shows**

*state-eq-trail*:  $S \sim T \Longrightarrow \text{trail } S = \text{trail } T$  **and**  
*state-eq-init-clss*:  $S \sim T \Longrightarrow \text{init-clss } S = \text{init-clss } T$  **and**  
*state-eq-learned-clss*:  $S \sim T \Longrightarrow \text{learned-clss } S = \text{learned-clss } T$  **and**

*state-eq-backtrack-lvl*:  $S \sim T \implies \text{backtrack-lvl } S = \text{backtrack-lvl } T$  **and**  
*state-eq-conflicting*:  $S \sim T \implies \text{conflicting } S = \text{conflicting } T$  **and**  
*state-eq-clauses*:  $S \sim T \implies \text{clauses } S = \text{clauses } T$  **and**  
*state-eq-undefined-lit*:  $S \sim T \implies \text{undefined-lit } (\text{trail } S) L = \text{undefined-lit } (\text{trail } T) L$   
 ⟨proof⟩

**lemma** *state-eq-conflicting-None*:

$S \sim T \implies \text{conflicting } T = \text{None} \implies \text{conflicting } S = \text{None}$   
 ⟨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  $\Rightarrow$  'st  $\Rightarrow$  'st **where**

*reduce-trail-to* F S =  
 (if length (trail S) = length F  $\vee$  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 = []  $\implies$  *reduce-trail-to* F S = S **and**  
*reduce-trail-to-eq-length*[simp]: length (trail S) = length F  $\implies$  *reduce-trail-to* F S = S  
 ⟨proof⟩

**lemma** *reduce-trail-to-length-ne*:

length (trail S)  $\neq$  length F  $\implies$  trail S  $\neq$  []  $\implies$   
*reduce-trail-to* F S = *reduce-trail-to* F (tl-trail S)  
 ⟨proof⟩

**lemma** *trail-reduce-trail-to-length-le*:

**assumes** length F > length (trail S)  
**shows** trail (reduce-trail-to F S) = []  
 ⟨proof⟩

**lemma** *trail-reduce-trail-to-Nil*[simp]:

trail (reduce-trail-to [] S) = []  
 ⟨proof⟩

**lemma** *clauses-reduce-trail-to-Nil*:

clauses (reduce-trail-to [] S) = clauses S  
 ⟨proof⟩

**lemma** *reduce-trail-to-skip-beginning*:

**assumes** trail S = F' @ F  
**shows** trail (reduce-trail-to F S) = F  
 ⟨proof⟩

**lemma** *clauses-reduce-trail-to[simp]*:  
*clauses* (reduce-trail-to  $F$   $S$ ) = *clauses*  $S$   
 ⟨proof⟩

**lemma** *conflicting-update-trail[simp]*:  
*conflicting* (reduce-trail-to  $F$   $S$ ) = *conflicting*  $S$   
 ⟨proof⟩

**lemma** *backtrack-lvl-update-trail[simp]*:  
*backtrack-lvl* (reduce-trail-to  $F$   $S$ ) = *backtrack-lvl*  $S$   
 ⟨proof⟩

**lemma** *init-clss-update-trail[simp]*:  
*init-clss* (reduce-trail-to  $F$   $S$ ) = *init-clss*  $S$   
 ⟨proof⟩

**lemma** *learned-clss-update-trail[simp]*:  
*learned-clss* (reduce-trail-to  $F$   $S$ ) = *learned-clss*  $S$   
 ⟨proof⟩

**lemma** *conflicting-reduce-trail-to[simp]*:  
*conflicting* (reduce-trail-to  $F$   $S$ ) = None  $\longleftrightarrow$  *conflicting*  $S$  = None  
 ⟨proof⟩

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

**lemma** *reduce-trail-to-state-eq<sub>NOT</sub>-compatible*:  
**assumes**  $ST$ :  $S \sim T$   
**shows** reduce-trail-to  $F$   $S \sim$  reduce-trail-to  $F$   $T$   
 ⟨proof⟩

**lemma** *reduce-trail-to-trail-tl-trail-decomp[simp]*:  
*trail*  $S$  =  $F' @ Decided\ K \# F \implies$  (*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*  $M'$   $\implies$  reduce-trail-to  $M$   $S$  = reduce-trail-to  $M'$   $S$   
 ⟨proof⟩

**lemma** *trail-reduce-trail-to-drop*:

*trail* (*reduce-trail-to* *F S*) =  
 (if *length* (*trail S*)  $\geq$  *length F*  
 then *drop* (*length* (*trail S*) - *length F*) (*trail S*)  
 else [])  
 <proof>

**lemma** *in-get-all-ann-decomposition-trail-update-trail*[*simp*]:

**assumes** *H*: (*L* # *M1*, *M2*)  $\in$  *set* (*get-all-ann-decomposition* (*trail S*))  
**shows** *trail* (*reduce-trail-to* *M1 S*) = *M1*  
 <proof>

**lemma** *conflicting-cons-trail-conflicting*[*simp*]:

**assumes** *undefined-lit* (*trail S*) (*lit-of L*)  
**shows**  
*conflicting* (*cons-trail L S*) = *None*  $\longleftrightarrow$  *conflicting S* = *None*  
 <proof>

**lemma** *conflicting-add-learned-cls-conflicting*[*simp*]:

*conflicting* (*add-learned-cls C S*) = *None*  $\longleftrightarrow$  *conflicting S* = *None*  
 <proof>

**lemma** *conflicting-update-backtrack-lvl*[*simp*]:

*conflicting* (*update-backtrack-lvl k S*) = *None*  $\longleftrightarrow$  *conflicting S* = *None*  
 <proof>

**end** — end of *state<sub>W</sub>* 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<sub>W</sub>* =

*state<sub>W</sub>*

— functions for the state:

— access functions:

*trail init-clss learned-clss backtrack-lvl conflicting*

— changing state:

*cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl*

*update-conflicting*

— get state:

*init-state*

**for**

*trail* :: '*st*  $\Rightarrow$  ('*v*, '*v* clause) *ann-lits* **and**

*init-clss* :: '*st*  $\Rightarrow$  '*v* clauses **and**

*learned-clss* :: '*st*  $\Rightarrow$  '*v* clauses **and**

*backtrack-lvl* :: '*st*  $\Rightarrow$  nat **and**

*conflicting* :: '*st*  $\Rightarrow$  '*v* clause option **and**

*cons-trail* :: ('*v*, '*v* clause) *ann-lit*  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* **and**

*tl-trail* :: '*st*  $\Rightarrow$  '*st* **and**

*add-learned-cls* :: '*v* clause  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* **and**

*remove-cls* :: '*v* clause  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* **and**

*update-backtrack-lvl* :: nat  $\Rightarrow$  '*st*  $\Rightarrow$  '*st* **and**

*update-conflicting* :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*init-state* :: 'v clauses  $\Rightarrow$  'st

**begin**

**inductive** *propagate* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for** *S* :: 'st **where**

*propagate-rule*: *conflicting S* = None  $\Rightarrow$

*E*  $\in \#$  clauses *S*  $\Rightarrow$

*L*  $\in \#$  *E*  $\Rightarrow$

*trail S*  $\models_{as} CNot (E - \{\#L\# \}) \Rightarrow$

*undefined-lit* (*trail S*) *L*  $\Rightarrow$

*T*  $\sim cons-trail$  (*Propagated L E*) *S*  $\Rightarrow$

*propagate S T*

**inductive-cases** *propagateE*: *propagate S T*

**inductive** *conflict* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for** *S* :: 'st **where**

*conflict-rule*:

*conflicting S* = None  $\Rightarrow$

*D*  $\in \#$  clauses *S*  $\Rightarrow$

*trail S*  $\models_{as} CNot D \Rightarrow$

*T*  $\sim update-conflicting$  (*Some D*) *S*  $\Rightarrow$

*conflict S T*

**inductive-cases** *conflictE*: *conflict S T*

**inductive** *backtrack* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for** *S* :: 'st **where**

*backtrack-rule*:

*conflicting S* = *Some D*  $\Rightarrow$

*L*  $\in \#$  *D*  $\Rightarrow$

(*Decided K*  $\#$  *M1*, *M2*)  $\in set (get-all-ann-decomposition (trail S)) \Rightarrow$

*get-level* (*trail S*) *L* = *backtrack-lvl S*  $\Rightarrow$

*get-level* (*trail S*) *L* = *get-maximum-level* (*trail S*) *D*  $\Rightarrow$

*get-maximum-level* (*trail S*) (*D* -  $\{\#L\# \}$ )  $\equiv i \Rightarrow$

*get-level* (*trail S*) *K* = *i* + 1  $\Rightarrow$

*T*  $\sim cons-trail$  (*Propagated L D*)

(*reduce-trail-to M1*

(*add-learned-cls D*

(*update-backtrack-lvl i*

(*update-conflicting None S*))))  $\Rightarrow$

*backtrack S T*

**inductive-cases** *backtrackE*: *backtrack S T*

**thm** *backtrackE*

**inductive** *decide* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for** *S* :: 'st **where**

*decide-rule*:

*conflicting S* = None  $\Rightarrow$

*undefined-lit* (*trail S*) *L*  $\Rightarrow$

*atm-of L*  $\in atms-of-mm (init-clss S) \Rightarrow$

*T*  $\sim cons-trail$  (*Decided L*) (*incr-lvl S*)  $\Rightarrow$

*decide S T*

**inductive-cases** *decideE*: *decide S T*

**inductive** *skip* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for** *S* :: 'st **where**

*skip-rule:*

$trail\ S = Propagated\ L\ C' \# M \implies$   
 $conflicting\ S = Some\ E \implies$   
 $-L \notin \# E \implies$   
 $E \neq \{\#\} \implies$   
 $T \sim tl-trail\ S \implies$   
 $skip\ S\ T$

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

*get-maximum-level* ( $Propagated\ L\ (C + \{\#L\#\}) \# M$ )  $D = k \vee k = 0$  (that was in a previous version of the book) is equivalent to *get-maximum-level* ( $Propagated\ L\ (C + \{\#L\#\}) \# M$ )  $D = k$ , when the structural invariants holds.

**inductive** *resolve* ::  $'st \Rightarrow 'st \Rightarrow bool$  **for**  $S :: 'st$  **where**

*resolve-rule*:  $trail\ S \neq [] \implies$

$hd-trail\ S = Propagated\ L\ E \implies$

$L \in \# E \implies$

$conflicting\ S = Some\ D' \implies$

$-L \in \# D' \implies$

$get-maximum-level\ (trail\ S)\ ((remove1-mset\ (-L)\ D')) = backtrack-lvl\ S \implies$

$T \sim update-conflicting\ (Some\ (resolve-cls\ L\ D'\ E))$

$(tl-trail\ S) \implies$

$resolve\ S\ T$

**inductive-cases** *resolveE*:  $resolve\ S\ T$

**inductive** *restart* ::  $'st \Rightarrow 'st \Rightarrow bool$  **for**  $S :: 'st$  **where**

*restart*:  $state\ S = (M, N, U, k, None) \implies$

$\neg M \models_{asm}\ clauses\ S \implies$

$U' \subseteq \# U \implies$

$state\ T = ([], N, U', 0, None) \implies$

$restart\ S\ T$

**inductive-cases** *restartE*:  $restart\ S\ T$

We add the condition  $C \notin \# init-clss\ S$ , to maintain consistency even without the strategy.

**inductive** *forget* ::  $'st \Rightarrow 'st \Rightarrow bool$  **where**

*forget-rule*:

$conflicting\ S = None \implies$

$C \in \# learned-clss\ S \implies$

$\neg(trail\ S) \models_{asm}\ clauses\ S \implies$

$C \notin set\ (get-all-mark-of-propagated\ (trail\ S)) \implies$

$C \notin \# init-clss\ S \implies$

$T \sim remove-cls\ C\ S \implies$

$forget\ S\ T$

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

**inductive** *cdcl<sub>W</sub>-rf* ::  $'st \Rightarrow 'st \Rightarrow bool$  **for**  $S :: 'st$  **where**

*restart*:  $restart\ S\ T \implies cdcl_W-rf\ S\ T \mid$

*forget*:  $forget\ S\ T \implies cdcl_W-rf\ S\ T$

**inductive** *cdcl<sub>W</sub>-bj* ::  $'st \Rightarrow 'st \Rightarrow bool$  **where**

*skip*:  $skip\ S\ S' \implies cdcl_W-bj\ S\ S' \mid$

*resolve*:  $resolve\ S\ S' \implies cdcl_W-bj\ S\ S' \mid$



*backtrack*:  $\text{backtrack } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S'$

**inductive-cases**  $\text{cdcl}_W\text{-bjE}$ :  $\text{cdcl}_W\text{-bj } S \ T$

**inductive**  $\text{cdcl}_W\text{-o} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

*decide*:  $\text{decide } S \ S' \implies \text{cdcl}_W\text{-o } S \ S' \mid$

*bj*:  $\text{cdcl}_W\text{-bj } S \ S' \implies \text{cdcl}_W\text{-o } S \ S'$

**inductive**  $\text{cdcl}_W :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

*propagate*:  $\text{propagate } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$

*conflict*:  $\text{conflict } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$

*other*:  $\text{cdcl}_W\text{-o } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$

*rf*:  $\text{cdcl}_W\text{-rf } S \ S' \implies \text{cdcl}_W \ S \ S'$

**lemma** *rtrancpl-propagate-is-rtrancpl-cdcl<sub>W</sub>*:

$\text{propagate}^{**} \ S \ S' \implies \text{cdcl}_W^{**} \ S \ S'$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-all-rules-induct}$ [*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

**fixes**  $S :: 'st$

**assumes**

$\text{cdcl}_W$ :  $\text{cdcl}_W \ S \ S'$  **and**

*propagate*:  $\bigwedge T. \text{propagate } S \ T \implies P \ S \ T$  **and**

*conflict*:  $\bigwedge T. \text{conflict } S \ T \implies P \ S \ T$  **and**

*forget*:  $\bigwedge T. \text{forget } S \ T \implies P \ S \ T$  **and**

*restart*:  $\bigwedge T. \text{restart } S \ T \implies P \ S \ T$  **and**

*decide*:  $\bigwedge T. \text{decide } S \ T \implies P \ S \ T$  **and**

*skip*:  $\bigwedge T. \text{skip } S \ T \implies P \ S \ T$  **and**

*resolve*:  $\bigwedge T. \text{resolve } S \ T \implies P \ S \ T$  **and**

*backtrack*:  $\bigwedge T. \text{backtrack } S \ T \implies P \ S \ T$

**shows**  $P \ S \ S'$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-all-induct}$ [*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

**fixes**  $S :: 'st$

**assumes**

$\text{cdcl}_W$ :  $\text{cdcl}_W \ S \ S'$  **and**

*propagateH*:  $\bigwedge C \ L \ T. \text{conflicting } S = \text{None} \implies$

$C \in \# \text{ clauses } S \implies$

$L \in \# \ C \implies$

$\text{trail } S \models_{\text{as}} C \text{Not } (\text{remove1-mset } L \ C) \implies$

$\text{undefined-lit } (\text{trail } S) \ L \implies$

$T \sim \text{cons-trail } (\text{Propagated } L \ C) \ S \implies$

$P \ S \ T$  **and**

*conflictH*:  $\bigwedge D \ T. \text{conflicting } S = \text{None} \implies$

$D \in \# \text{ clauses } S \implies$

$\text{trail } S \models_{\text{as}} C \text{Not } D \implies$

$T \sim \text{update-conflicting } (\text{Some } D) \ S \implies$

$P \ S \ T$  **and**

*forgetH*:  $\bigwedge C \ T. \text{conflicting } S = \text{None} \implies$

$C \in \# \text{ learned-clss } S \implies$

$\neg(\text{trail } S) \models_{\text{asm}} \text{clauses } S \implies$

$C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \implies$

$C \notin \# \text{ init-clss } S \implies$

$T \sim \text{remove-cls } C \ S \implies$   
 $P \ S \ T \text{ and}$   
 $\text{restartH: } \bigwedge T \ U. \neg \text{trail } S \models \text{asm clauses } S \implies$   
 $\text{conflicting } S = \text{None} \implies$   
 $\text{state } T = ([], \text{init-clss } S, U, 0, \text{None}) \implies$   
 $U \subseteq \# \text{ learned-clss } S \implies$   
 $P \ S \ T \text{ and}$   
 $\text{decideH: } \bigwedge L \ T. \text{ conflicting } S = \text{None} \implies$   
 $\text{undefined-lit } (\text{trail } S) \ L \implies$   
 $\text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S) \implies$   
 $T \sim \text{cons-trail } (\text{Decided } L) \ (\text{incr-lvl } S) \implies$   
 $P \ S \ T \text{ and}$   
 $\text{skipH: } \bigwedge L \ C' \ M \ E \ T.$   
 $\text{trail } S = \text{Propagated } L \ C' \ \# \ M \implies$   
 $\text{conflicting } S = \text{Some } E \implies$   
 $-L \notin \# \ E \implies E \neq \{\#\} \implies$   
 $T \sim \text{tl-trail } S \implies$   
 $P \ S \ T \text{ and}$   
 $\text{resolveH: } \bigwedge L \ E \ M \ D \ T.$   
 $\text{trail } S = \text{Propagated } L \ E \ \# \ M \implies$   
 $L \in \# \ E \implies$   
 $\text{hd-trail } S = \text{Propagated } L \ E \implies$   
 $\text{conflicting } S = \text{Some } D \implies$   
 $-L \in \# \ D \implies$   
 $\text{get-maximum-level } (\text{trail } S) \ ((\text{remove1-mset } (-L) \ D)) = \text{backtrack-lvl } S \implies$   
 $T \sim \text{update-conflicting}$   
 $(\text{Some } (\text{resolve-cls } L \ D \ E)) \ (\text{tl-trail } S) \implies$   
 $P \ S \ T \text{ and}$   
 $\text{backtrackH: } \bigwedge L \ D \ K \ i \ M1 \ M2 \ T.$   
 $\text{conflicting } S = \text{Some } D \implies$   
 $L \in \# \ D \implies$   
 $(\text{Decided } K \ \# \ M1, \ M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S)) \implies$   
 $\text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S \implies$   
 $\text{get-level } (\text{trail } S) \ L = \text{get-maximum-level } (\text{trail } S) \ D \implies$   
 $\text{get-maximum-level } (\text{trail } S) \ (\text{remove1-mset } L \ D) \equiv i \implies$   
 $\text{get-level } (\text{trail } S) \ K = i+1 \implies$   
 $T \sim \text{cons-trail } (\text{Propagated } L \ D)$   
 $(\text{reduce-trail-to } M1$   
 $(\text{add-learned-cls } D$   
 $(\text{update-backtrack-lvl } i$   
 $(\text{update-conflicting } \text{None } S)))) \implies$   
 $P \ S \ T$   
**shows**  $P \ S \ S'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-o-induct}[\text{consumes } 1, \text{ case-names decide skip resolve backtrack}]:$

**fixes**  $S :: 'st$

**assumes**  $\text{cdcl}_W: \text{cdcl}_W\text{-o } S \ T \text{ and}$

$\text{decideH: } \bigwedge L \ T. \text{ conflicting } S = \text{None} \implies \text{undefined-lit } (\text{trail } S) \ L$   
 $\implies \text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S)$   
 $\implies T \sim \text{cons-trail } (\text{Decided } L) \ (\text{incr-lvl } S)$   
 $\implies P \ S \ T \text{ and}$

$\text{skipH: } \bigwedge L \ C' \ M \ E \ T.$

$\text{trail } S = \text{Propagated } L \ C' \ \# \ M \implies$   
 $\text{conflicting } S = \text{Some } E \implies$   
 $-L \notin \# \ E \implies E \neq \{\#\} \implies$

$T \sim \text{tl-trail } S \implies$   
 $P \ S \ T \text{ and}$   
 $\text{resolveH: } \bigwedge L \ E \ M \ D \ T.$   
 $\text{trail } S = \text{Propagated } L \ E \ \# \ M \implies$   
 $L \in \# \ E \implies$   
 $\text{hd-trail } S = \text{Propagated } L \ E \implies$   
 $\text{conflicting } S = \text{Some } D \implies$   
 $-L \in \# \ D \implies$   
 $\text{get-maximum-level } (\text{trail } S) ((\text{remove1-mset } (-L) \ D)) = \text{backtrack-lvl } S \implies$   
 $T \sim \text{update-conflicting}$   
 $(\text{Some } (\text{resolve-cls } L \ D \ E)) (\text{tl-trail } S) \implies$   
 $P \ S \ T \text{ and}$   
 $\text{backtrackH: } \bigwedge L \ D \ K \ i \ M1 \ M2 \ T.$   
 $\text{conflicting } S = \text{Some } D \implies$   
 $L \in \# \ D \implies$   
 $(\text{Decided } K \ \# \ M1, \ M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S)) \implies$   
 $\text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S \implies$   
 $\text{get-level } (\text{trail } S) \ L = \text{get-maximum-level } (\text{trail } S) \ D \implies$   
 $\text{get-maximum-level } (\text{trail } S) (\text{remove1-mset } L \ D) \equiv i \implies$   
 $\text{get-level } (\text{trail } S) \ K = i + 1 \implies$   
 $T \sim \text{cons-trail } (\text{Propagated } L \ D)$   
 $(\text{reduce-trail-to } M1$   
 $(\text{add-learned-cls } D$   
 $(\text{update-backtrack-lvl } i$   
 $(\text{update-conflicting } \text{None } S)))) \implies$   
 $P \ S \ T$   
**shows**  $P \ S \ T$   
 $\langle \text{proof} \rangle$

**thm**  $\text{cdcl}_W\text{-o.induct}$

**lemma**  $\text{cdcl}_W\text{-o.all-rules-induct}[\text{consumes } 1, \text{ case-names decide backtrack skip resolve}]$ :

**fixes**  $S \ T :: 'st$

**assumes**

$\text{cdcl}_W\text{-o } S \ T \text{ and}$

$\bigwedge T. \text{decide } S \ T \implies P \ S \ T \text{ and}$

$\bigwedge T. \text{backtrack } S \ T \implies P \ S \ T \text{ and}$

$\bigwedge T. \text{skip } S \ T \implies P \ S \ T \text{ and}$

$\bigwedge T. \text{resolve } S \ T \implies P \ S \ T$

**shows**  $P \ S \ T$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-o.rule-cases}[\text{consumes } 1, \text{ case-names decide backtrack skip resolve}]$ :

**fixes**  $S \ T :: 'st$

**assumes**

$\text{cdcl}_W\text{-o } S \ T \text{ and}$

$\text{decide } S \ T \implies P \text{ and}$

$\text{backtrack } S \ T \implies P \text{ and}$

$\text{skip } S \ T \implies P \text{ and}$

$\text{resolve } S \ T \implies P$

**shows**  $P$

$\langle \text{proof} \rangle$

### 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-skipped*:

**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<sub>W</sub>-distinctinv-1*:

**assumes**

*cdcl<sub>W</sub> 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<sub>W</sub>-consistent-inv-2*:

**assumes**

*cdcl<sub>W</sub> 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<sub>W</sub>-o-bt*:

**assumes**

*cdcl<sub>W-o</sub> 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<sub>W</sub>-rf-bt*:

**assumes**

*cdcl<sub>W-rf</sub> 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<sub>W</sub>-bt*:

**assumes**

$cdcl_W S S'$  **and**  
 $backtrack\_lvl S = count\_decided (trail S)$  **and**  
 $no\_dup (trail S)$   
**shows**  $backtrack\_lvl S' = count\_decided (trail S')$   
 $\langle proof \rangle$

We write  $1 + count\_decided (trail S)$  instead of  $backtrack\_lvl S$  to avoid non termination of rewriting.

**definition**  $cdcl_W\text{-}M\text{-level-inv} :: 'st \Rightarrow bool$  **where**  
 $cdcl_W\text{-}M\text{-level-inv} S \iff$   
 $consistent\_interp (lits\_of\_l (trail S))$   
 $\wedge no\_dup (trail S)$   
 $\wedge backtrack\_lvl S = count\_decided (trail S)$

**lemma**  $cdcl_W\text{-}M\text{-level-inv-decomp}$ :  
**assumes**  $cdcl_W\text{-}M\text{-level-inv} S$   
**shows**  
 $consistent\_interp (lits\_of\_l (trail S))$  **and**  
 $no\_dup (trail S)$   
 $\langle proof \rangle$

**lemma**  $cdcl_W\text{-consistent-inv}$ :  
**fixes**  $S S' :: 'st$   
**assumes**  
 $cdcl_W S S'$  **and**  
 $cdcl_W\text{-}M\text{-level-inv} S$   
**shows**  $cdcl_W\text{-}M\text{-level-inv} S'$   
 $\langle proof \rangle$

**lemma**  $rtrancp\text{-}cdcl_W\text{-consistent-inv}$ :  
**assumes**  
 $cdcl_W^{**} S S'$  **and**  
 $cdcl_W\text{-}M\text{-level-inv} S$   
**shows**  $cdcl_W\text{-}M\text{-level-inv} S'$   
 $\langle proof \rangle$

**lemma**  $trancp\text{-}cdcl_W\text{-consistent-inv}$ :  
**assumes**  
 $cdcl_W^{++} S S'$  **and**  
 $cdcl_W\text{-}M\text{-level-inv} S$   
**shows**  $cdcl_W\text{-}M\text{-level-inv} S'$   
 $\langle proof \rangle$

**lemma**  $cdcl_W\text{-}M\text{-level-inv}\text{-}S0\text{-}cdcl_W[simp]$ :  
 $cdcl_W\text{-}M\text{-level-inv} (init\_state N)$   
 $\langle proof \rangle$

**lemma**  $cdcl_W\text{-}M\text{-level-inv}\text{-}get\_level\text{-}le\text{-}backtrack\_lvl$ :  
**assumes**  $inv: cdcl_W\text{-}M\text{-level-inv} S$   
**shows**  $get\_level (trail S) L \leq backtrack\_lvl S$   
 $\langle proof \rangle$

**lemma**  $backtrack\text{-}ex\text{-}decomp$ :  
**assumes**  
 $M\text{-}l: cdcl_W\text{-}M\text{-level-inv} S$  **and**  
 $i\text{-}S: i < backtrack\_lvl S$

**shows**  $\exists K M1 M2. (Decided K \# M1, M2) \in set (get-all-ann-decomposition (trail S)) \wedge$   
 $get-level (trail S) K = Suc i$   
 $\langle proof \rangle$

**lemma** *backtrack-lvl-backtrack-decrease*:  
**assumes** *inv*: *cdcl<sub>W</sub>-M-level-inv S* **and** *bt*: *backtrack S T*  
**shows** *backtrack-lvl T < backtrack-lvl S*  
 $\langle proof \rangle$

### Compatibility with $op \sim$

**lemma** *propagate-state-eq-compatible*:  
**assumes**  
*propa*: *propagate S T* **and**  
*SS'*:  $S \sim S'$  **and**  
*TT'*:  $T \sim T'$   
**shows** *propagate S' T'*  
 $\langle proof \rangle$

**lemma** *conflict-state-eq-compatible*:  
**assumes**  
*conf*: *conflict S T* **and**  
*TT'*:  $T \sim T'$  **and**  
*SS'*:  $S \sim S'$   
**shows** *conflict S' T'*  
 $\langle proof \rangle$

**lemma** *backtrack-state-eq-compatible*:  
**assumes**  
*bt*: *backtrack S T* **and**  
*SS'*:  $S \sim S'$  **and**  
*TT'*:  $T \sim T'$  **and**  
*inv*: *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *backtrack S' T'*  
 $\langle proof \rangle$

**lemma** *decide-state-eq-compatible*:  
**assumes**  
*decide* *S T* **and**  
 $S \sim S'$  **and**  
 $T \sim T'$   
**shows** *decide S' T'*  
 $\langle proof \rangle$

**lemma** *skip-state-eq-compatible*:  
**assumes**  
*skip*: *skip S T* **and**  
*SS'*:  $S \sim S'$  **and**  
*TT'*:  $T \sim T'$   
**shows** *skip S' T'*  
 $\langle proof \rangle$

**lemma** *resolve-state-eq-compatible*:  
**assumes**  
*res*: *resolve S T* **and**  
*TT'*:  $T \sim T'$  **and**

$SS': S \sim S'$   
**shows** *resolve*  $S' T'$   
 $\langle \text{proof} \rangle$

**lemma** *forget-state-eq-compatible:*

**assumes**  
*forget: forget*  $S T$  **and**  
 $SS': S \sim S'$  **and**  
 $TT': T \sim T'$   
**shows** *forget*  $S' T'$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-state-eq-compatible:*

**assumes**  
 $cdcl_W S T$  **and**  $\neg \text{restart } S T$  **and**  
 $S \sim S'$   
 $T \sim T'$  **and**  
 $cdcl_W\text{-}M\text{-level-inv } S$   
**shows**  $cdcl_W S' T'$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-bj-state-eq-compatible:*

**assumes**  
 $cdcl_W\text{-}bj S T$  **and**  $cdcl_W\text{-}M\text{-level-inv } S$   
 $T \sim T'$   
**shows**  $cdcl_W\text{-}bj S T'$   
 $\langle \text{proof} \rangle$

**lemma** *trancpl-cdcl<sub>W</sub>-bj-state-eq-compatible:*

**assumes**  
 $cdcl_W\text{-}bj^{++} S T$  **and**  $inv: cdcl_W\text{-}M\text{-level-inv } S$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$   
**shows**  $cdcl_W\text{-}bj^{++} S' T'$   
 $\langle \text{proof} \rangle$

## Conservation of some Properties

**lemma** *cdcl<sub>W</sub>-o-no-more-init-clss:*

**assumes**  
 $cdcl_W\text{-}o S S'$  **and**  
 $inv: cdcl_W\text{-}M\text{-level-inv } S$   
**shows**  $init\text{-}clss S = init\text{-}clss S'$   
 $\langle \text{proof} \rangle$

**lemma** *trancpl-cdcl<sub>W</sub>-o-no-more-init-clss:*

**assumes**  
 $cdcl_W\text{-}o^{++} S S'$  **and**  
 $inv: cdcl_W\text{-}M\text{-level-inv } S$   
**shows**  $init\text{-}clss S = init\text{-}clss S'$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>W</sub>-o-no-more-init-clss:*

**assumes**  
 $cdcl_W\text{-}o^{**} S S'$  **and**  
 $inv: cdcl_W\text{-}M\text{-level-inv } S$

**shows**  $init-clss\ S = init-clss\ S'$   
 $\langle proof \rangle$

**lemma**  $cdcl_W$ -init-clss:

**assumes**  
 $cdcl_W\ S\ T$  **and**  
 $inv: cdcl_W$ -M-level-inv  $S$   
**shows**  $init-clss\ S = init-clss\ T$   
 $\langle proof \rangle$

**lemma**  $rtrancp$ - $cdcl_W$ -init-clss:

$cdcl_W^{**}\ S\ T \implies cdcl_W$ -M-level-inv  $S \implies init-clss\ S = init-clss\ T$   
 $\langle proof \rangle$

**lemma**  $trancp$ - $cdcl_W$ -init-clss:

$cdcl_W^{++}\ S\ T \implies cdcl_W$ -M-level-inv  $S \implies init-clss\ S = init-clss\ T$   
 $\langle proof \rangle$

## 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  $(S :: 'st) \longleftrightarrow$

$(init-clss\ S \models_{psm} learned-clss\ S$   
 $\wedge (\forall T. conflicting\ S = Some\ T \longrightarrow init-clss\ S \models_{pm}\ T)$   
 $\wedge set\ (get-all-mark-of-propagated\ (trail\ S)) \subseteq set-mset\ (clauses\ S))$

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

**lemma**  $cdcl_W$ -learned-clause-S0- $cdcl_W$ [simp]:

$cdcl_W$ -learned-clause  $(init-state\ N)$   
 $\langle proof \rangle$

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  $S'$   
 $\langle proof \rangle$

**lemma**  $rtrancp$ - $cdcl_W$ -learned-clss:

**assumes**  
 $cdcl_W^{**}\ S\ S'$  **and**  
 $cdcl_W$ -M-level-inv  $S$   
 $cdcl_W$ -learned-clause  $S$   
**shows**  $cdcl_W$ -learned-clause  $S'$   
 $\langle proof \rangle$



## No alien atom in the state

This invariant means that all the literals are in the set of clauses. These properties are implicit in Weidenbach's book.

**definition** *no-strange-atm*  $S' \longleftrightarrow$  (

( $\forall T$ . *conflicting*  $S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S')$ )  
 $\wedge$  ( $\forall L$  *mark*. *Propagated*  $L$  *mark*  $\in \text{set } (\text{trail } S')$   
 $\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S')$ )  
 $\wedge \text{atms-of-mm } (\text{learned-clss } S') \subseteq \text{atms-of-mm } (\text{init-clss } S')$   
 $\wedge \text{atm-of ' } (\text{lits-of-l } (\text{trail } S')) \subseteq \text{atms-of-mm } (\text{init-clss } S')$ )

**lemma** *no-strange-atm-decomp*:

**assumes** *no-strange-atm*  $S$

**shows** *conflicting*  $S = \text{Some } T \implies \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S)$

**and** ( $\forall L$  *mark*. *Propagated*  $L$  *mark*  $\in \text{set } (\text{trail } S)$

$\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S)$ )

**and**  $\text{atms-of-mm } (\text{learned-clss } S) \subseteq \text{atms-of-mm } (\text{init-clss } S)$

**and**  $\text{atm-of ' } (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{init-clss } S)$

$\langle \text{proof} \rangle$

**lemma** *no-strange-atm-S0* [*simp*]: *no-strange-atm* (*init-state*  $N$ )

$\langle \text{proof} \rangle$

**lemma** *in-atms-of-implies-atm-of-on-atms-of-ms*:

$C + \{\#L\# \} \in \# A \implies x \in \text{atms-of } C \implies x \in \text{atms-of-mm } A$

$\langle \text{proof} \rangle$

**lemma** *propagate-no-strange-atm-inv*:

**assumes**

*propagate*  $S$   $T$  **and**

*alien*: *no-strange-atm*  $S$

**shows** *no-strange-atm*  $T$

$\langle \text{proof} \rangle$

**lemma** *in-atms-of-remove1-mset-in-atms-of*:

$x \in \text{atms-of } (\text{remove1-mset } L C) \implies x \in \text{atms-of } C$

$\langle \text{proof} \rangle$

**lemma** *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*:

$\text{atms-of-mm } (\text{learned-clss } S) \subseteq \text{atms-of-mm } (\text{init-clss } S) \implies$

$x \in \text{atms-of-mm } (\text{learned-clss } T) \implies$

$\text{learned-clss } T \subseteq \# \text{ learned-clss } S \implies$

$x \in \text{atms-of-mm } (\text{init-clss } S)$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-no-strange-atm-explicit*:

**assumes**

*cdcl<sub>W</sub>*  $S$   $S'$  **and**

*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**

*conf*:  $\forall T$ . *conflicting*  $S = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S)$  **and**

*decided*:  $\forall L$  *mark*. *Propagated*  $L$  *mark*  $\in \text{set } (\text{trail } S)$

$\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S)$  **and**

*learned*:  $\text{atms-of-mm } (\text{learned-clss } S) \subseteq \text{atms-of-mm } (\text{init-clss } S)$  **and**

*trail*:  $\text{atm-of ' } (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{init-clss } S)$

**shows**

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

**lemma** *cdcl<sub>W</sub>-no-strange-atm-inv:*

**assumes** *cdcl<sub>W</sub> S S' and no-strange-atm S and cdcl<sub>W</sub>-M-level-inv S*  
**shows** *no-strange-atm S'*  
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-no-strange-atm-inv:*

**assumes** *cdcl<sub>W</sub>\*\* S S' and no-strange-atm S and cdcl<sub>W</sub>-M-level-inv S*  
**shows** *no-strange-atm S'*  
 $\langle \text{proof} \rangle$

## No Duplicates all Around

This invariant shows that there is no duplicate (no literal appearing twice in the formula). The last part could be proven using the previous invariant also. Remark that we will show later that there cannot be duplicate *clause*.

**definition** *distinct-cdcl<sub>W</sub>-state (S :: 'st)*

$\longleftrightarrow ((\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{distinct-mset } T)$   
 $\wedge \text{distinct-mset-mset } (\text{learned-clss } S)$   
 $\wedge \text{distinct-mset-mset } (\text{init-clss } S)$   
 $\wedge (\forall L \text{ mark. (Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset mark})))$

**lemma** *distinct-cdcl<sub>W</sub>-state-decomp:*

**assumes** *distinct-cdcl<sub>W</sub>-state (S :: 'st)*  
**shows**  
 $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{distinct-mset } T$  **and**  
 $\text{distinct-mset-mset } (\text{learned-clss } S)$  **and**  
 $\text{distinct-mset-mset } (\text{init-clss } S)$  **and**  
 $\forall L \text{ mark. (Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset mark})$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-cdcl<sub>W</sub>-state-decomp-2:*

**assumes** *distinct-cdcl<sub>W</sub>-state (S :: 'st) and conflicting S = Some T*  
**shows** *distinct-mset T*  
 $\langle \text{proof} \rangle$

**lemma** *distinct-cdcl<sub>W</sub>-state-S0-cdcl<sub>W</sub>[simp]:*

$\text{distinct-mset-mset } N \implies \text{distinct-cdcl}_W\text{-state } (\text{init-state } N)$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-cdcl<sub>W</sub>-state-inv:*

**assumes**  
 $\text{cdcl}_W S S'$  **and**  
 $\text{lev-inv: cdcl}_W\text{-M-level-inv } S$  **and**  
 $\text{distinct-cdcl}_W\text{-state } S$   
**shows** *distinct-cdcl<sub>W</sub>-state S'*  
 $\langle \text{proof} \rangle$

**lemma** *rtanclp-distinct-cdcl<sub>W</sub>-state-inv*:

**assumes**  
 $cdcl_W^{**} S S'$  **and**  
 $cdcl_W\text{-}M\text{-level-inv } S$  **and**  
 $distinct\text{-}cdcl_W\text{-}state S$   
**shows**  $distinct\text{-}cdcl_W\text{-}state S'$   
 $\langle proof \rangle$

## 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* ::  $'st \Rightarrow bool$  **where**

$every\text{-}mark\text{-}is\text{-}a\text{-}conflict S \equiv$   
 $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark} \# b = (trail S)$   
 $\longrightarrow (b \models_{as} CNot (mark - \{\#L\}) \wedge L \in \# \text{ mark})$

**definition** *cdcl<sub>W</sub>-conflicting*  $S \longleftrightarrow$

$(\forall T. \text{conflicting } S = \text{Some } T \longrightarrow trail S \models_{as} CNot T)$   
 $\wedge every\text{-}mark\text{-}is\text{-}a\text{-}conflict S$

**lemma** *backtrack-atms-of-D-in-M1*:

**fixes**  $M1 :: ('v, 'v \text{ clause}) \text{ ann-lits}$   
**assumes**  
 $inv: cdcl_W\text{-}M\text{-level-inv } S$  **and**  
 $i: \text{get-maximum-level } (trail S) ((\text{remove1-mset } L D)) \equiv i$  **and**  
 $decomp: (\text{Decided } K \# M1, M2)$   
 $\in \text{set } (\text{get-all-ann-decomposition } (trail S))$  **and**  
 $S\text{-lvl}: \text{backtrack-lvl } S = \text{get-maximum-level } (trail S) D$  **and**  
 $S\text{-confl}: \text{conflicting } S = \text{Some } D$  **and**  
 $lev\text{-}K: \text{get-level } (trail S) K = \text{Suc } i$  **and**  
 $T: T \sim \text{cons-trail } (\text{Propagated } L D)$   
 $(\text{reduce-trail-to } M1$   
 $(\text{add-learned-cls } D$   
 $(\text{update-backtrack-lvl } i$   
 $(\text{update-conflicting } None S))))$  **and**  
 $confl: \forall T. \text{conflicting } S = \text{Some } T \longrightarrow trail S \models_{as} CNot T$   
**shows**  $\text{atms-of } ((\text{remove1-mset } L D)) \subseteq \text{atm-of } ' \text{ lits-of-l } (tl (trail T))$   
 $\langle proof \rangle$

**lemma** *distinct-atms-of-incl-not-in-other*:

**assumes**  
 $a1: \text{no-dup } (M @ M')$  **and**  
 $a2: \text{atms-of } D \subseteq \text{atm-of } ' \text{ lits-of-l } M'$  **and**  
 $a3: x \in \text{atms-of } D$   
**shows**  $x \notin \text{atm-of } ' \text{ lits-of-l } M$   
 $\langle proof \rangle$

Item 5 page 81 of Weidenbach's book

**lemma** *cdcl<sub>W</sub>-propagate-is-conclusion*:

**assumes**  
 $cdcl_W S S'$  **and**  
 $inv: cdcl_W\text{-}M\text{-level-inv } S$  **and**  
 $decomp: \text{all-decomposition-implies-m } (\text{init-clss } S) (\text{get-all-ann-decomposition } (trail S))$  **and**  
 $\text{learned}: cdcl_W\text{-learned-clause } S$  **and**

*conft*:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$  **and**  
*alien*: *no-strange-atm*  $S$   
**shows** *all-decomposition-implies-m* (*init-clss*  $S'$ ) (*get-all-ann-decomposition* (*trail*  $S'$ ))  
 <proof>

**lemma** *cdcl<sub>W</sub>-propagate-is-false*:

**assumes**  
*cdcl<sub>W</sub>*  $S$   $S'$  **and**  
*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**  
*learned*: *cdcl<sub>W</sub>-learned-clause*  $S$  **and**  
*decomp*: *all-decomposition-implies-m* (*init-clss*  $S$ ) (*get-all-ann-decomposition* (*trail*  $S$ )) **and**  
*conft*:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$  **and**  
*alien*: *no-strange-atm*  $S$  **and**  
*mark-conft*: *every-mark-is-a-conflict*  $S$   
**shows** *every-mark-is-a-conflict*  $S'$   
 <proof>

**lemma** *cdcl<sub>W</sub>-conflicting-is-false*:

**assumes**  
*cdcl<sub>W</sub>*  $S$   $S'$  **and**  
*M-lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**  
*conft-inv*:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$  **and**  
*decided-conft*:  $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$   
 $\longrightarrow (b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$  **and**  
*dist*: *distinct-cdcl<sub>W</sub>-state*  $S$   
**shows**  $\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{trail } S' \models_{as} \text{CNot } T$   
 <proof>

**lemma** *cdcl<sub>W</sub>-conflicting-decomp*:

**assumes** *cdcl<sub>W</sub>-conflicting*  $S$   
**shows**  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$   
**and**  $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$   
 $\longrightarrow (b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$   
 <proof>

**lemma** *cdcl<sub>W</sub>-conflicting-decomp2*:

**assumes** *cdcl<sub>W</sub>-conflicting*  $S$  **and** *conflicting*  $S = \text{Some } T$   
**shows**  $\text{trail } S \models_{as} \text{CNot } T$   
 <proof>

**lemma** *cdcl<sub>W</sub>-conflicting-S0-cdcl<sub>W</sub>[simp]*:

*cdcl<sub>W</sub>-conflicting* (*init-state*  $N$ )  
 <proof>

## Putting all the invariants together

**lemma** *cdcl<sub>W</sub>-all-inv*:

**assumes**  
*cdcl<sub>W</sub>*: *cdcl<sub>W</sub>*  $S$   $S'$  **and**  
 1: *all-decomposition-implies-m* (*init-clss*  $S$ ) (*get-all-ann-decomposition* (*trail*  $S$ )) **and**  
 2: *cdcl<sub>W</sub>-learned-clause*  $S$  **and**  
 4: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**  
 5: *no-strange-atm*  $S$  **and**  
 7: *distinct-cdcl<sub>W</sub>-state*  $S$  **and**  
 8: *cdcl<sub>W</sub>-conflicting*  $S$   
**shows**

*all-decomposition-implies-m* (*init-clss*  $S'$ ) (*get-all-ann-decomposition* (*trail*  $S'$ )) **and**  
*cdcl<sub>W</sub>-learned-clause*  $S'$  **and**  
*cdcl<sub>W</sub>-M-level-inv*  $S'$  **and**  
*no-strange-atm*  $S'$  **and**  
*distinct-cdcl<sub>W</sub>-state*  $S'$  **and**  
*cdcl<sub>W</sub>-conflicting*  $S'$   
 ⟨*proof*⟩

**lemma** *rtrancp-cdcl<sub>W</sub>-all-inv*:

**assumes**

*cdcl<sub>W</sub>*: *rtrancp cdcl<sub>W</sub> S S'* **and**

1: *all-decomposition-implies-m* (*init-clss*  $S$ ) (*get-all-ann-decomposition* (*trail*  $S$ )) **and**

2: *cdcl<sub>W</sub>-learned-clause*  $S$  **and**

4: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**

5: *no-strange-atm*  $S$  **and**

7: *distinct-cdcl<sub>W</sub>-state*  $S$  **and**

8: *cdcl<sub>W</sub>-conflicting*  $S$

**shows**

*all-decomposition-implies-m* (*init-clss*  $S'$ ) (*get-all-ann-decomposition* (*trail*  $S'$ )) **and**

*cdcl<sub>W</sub>-learned-clause*  $S'$  **and**

*cdcl<sub>W</sub>-M-level-inv*  $S'$  **and**

*no-strange-atm*  $S'$  **and**

*distinct-cdcl<sub>W</sub>-state*  $S'$  **and**

*cdcl<sub>W</sub>-conflicting*  $S'$

⟨*proof*⟩

**lemma** *all-invariant-S0-cdcl<sub>W</sub>*:

**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<sub>W</sub>-learned-clause* (*init-state*  $N$ ) **and**

$\forall T$ . *conflicting* (*init-state*  $N$ ) = *Some*  $T \longrightarrow$  (*trail* (*init-state*  $N$ ))  $\models_{as}$  *CNot*  $T$  **and**

*no-strange-atm* (*init-state*  $N$ ) **and**

*consistent-interp* (*lits-of-l* (*trail* (*init-state*  $N$ ))) **and**

$\forall L$  *mark*  $a$ .  $a \text{ @ } \text{Propagated } L \text{ mark } \# b = \text{trail } (\text{init-state } N) \longrightarrow$

$(b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{mark})$  **and**

*distinct-cdcl<sub>W</sub>-state* (*init-state*  $N$ )

⟨*proof*⟩

Item 6 page 81 of Weidenbach's book

**lemma** *cdcl<sub>W</sub>-only-propagated-vars-unsat*:

**assumes**

*decided*:  $\forall x \in \text{set } M. \neg \text{is-decided } x$  **and**

*DN*:  $D \in \# \text{ clauses } S$  **and**

*D*:  $M \models_{as} \text{CNot } D$  **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<sub>W</sub>-learned-clause*  $S$  **and**

*atm-incl*: *no-strange-atm*  $S$

**shows** *unsatisfiable* (*set-mset*  $N$ )

⟨*proof*⟩

Item 5 page 81 of Weidenbach's book

We have actually a much stronger theorem, namely *all-decomposition-implies-propagated-lits-are-implied*,

that show that the only choices we made are decided in the formula

**lemma**

**assumes** *all-decomposition-implies-m*  $N$  (*get-all-ann-decomposition*  $M$ )  
**and**  $\forall m \in \text{set } M. \neg \text{is-decided } m$   
**shows**  $\text{set-mset } N \models_{ps} \text{unmark-l } M$   
 $\langle \text{proof} \rangle$

Item 7 page 81 of Weidenbach's book (part 1)

**lemma** *conflict-with-false-implies-unsat*:

**assumes**  
 $\text{cdcl}_W: \text{cdcl}_W \ S \ S'$  **and**  
 $\text{lev}: \text{cdcl}_W\text{-}M\text{-level-inv } S$  **and**  
 $[\text{simp}]: \text{conflicting } S' = \text{Some } \{\#\}$  **and**  
 $\text{learned}: \text{cdcl}_W\text{-learned-clause } S$   
**shows**  $\text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$   
 $\langle \text{proof} \rangle$

Item 7 page 81 of Weidenbach's book (part 2)

**lemma** *conflict-with-false-implies-terminated*:

**assumes**  $\text{cdcl}_W \ S \ S'$   
**and**  $\text{conflicting } S = \text{Some } \{\#\}$   
**shows**  $\text{False}$   
 $\langle \text{proof} \rangle$

## 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**  
 $\text{cdcl}_W \ S \ S'$  **and**  
 $\text{lev}: \text{cdcl}_W\text{-}M\text{-level-inv } S$  **and**  
 $\text{conflicting}: \text{cdcl}_W\text{-conflicting } S$  **and**  
 $\text{no-tauto}: \forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$   
**shows**  $\forall s \in \# \text{ learned-clss } S'. \neg \text{tautology } s$   
 $\langle \text{proof} \rangle$

**definition** *final-cdcl<sub>W</sub>-state* ( $S :: 'st$ )

$\longleftrightarrow (\text{trail } S \models_{asm} \text{init-clss } S$   
 $\vee ((\forall L \in \text{set } (\text{trail } S). \neg \text{is-decided } L) \wedge$   
 $(\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} \text{CNot } C)))$

**definition** *termination-cdcl<sub>W</sub>-state* ( $S :: 'st$ )

$\longleftrightarrow (\text{trail } S \models_{asm} \text{init-clss } S$   
 $\vee ((\forall L \in \text{atms-of-mm } (\text{init-clss } S). L \in \text{atm-of ' lits-of-l } (\text{trail } S))$   
 $\wedge (\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} \text{CNot } C)))$

### 6.1.4 CDCL Strong Completeness

**lemma** *cdcl<sub>W</sub>-can-do-step*:

**assumes**  
 $\text{consistent-interp } (\text{set } M)$  **and**  
 $\text{distinct } M$  **and**  
 $\text{atm-of ' } (\text{set } M) \subseteq \text{atms-of-mm } N$

**shows**  $\exists S. \text{rtrancpl } \text{cdcl}_W \text{ (init-state } N) S$   
 $\wedge \text{state } S = (\text{map } (\lambda L. \text{Decided } L) M, N, \{\#\}, \text{length } M, \text{None})$   
 $\langle \text{proof} \rangle$

theorem 2.9.11 page 84 of Weidenbach's book

**lemma** *cdcl<sub>W</sub>-strong-completeness*:

**assumes**

*MN*: *set*  $M \models_{sm} N$  **and**  
*cons*: *consistent-interp* (*set*  $M$ ) **and**  
*dist*: *distinct*  $M$  **and**  
*atm*: *atm-of* ' (*set*  $M$ )  $\subseteq$  *atms-of-mm*  $N$

**obtains**  $S$  **where**

*state*  $S = (\text{map } (\lambda L. \text{Decided } L) M, N, \{\#\}, \text{length } M, \text{None})$  **and**  
 $\text{rtrancpl } \text{cdcl}_W \text{ (init-state } N) S$  **and**  
*final-cdcl<sub>W</sub>-state*  $S$

$\langle \text{proof} \rangle$

### 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** *trancpl-conflict*:

*trancpl conflict*  $S S' \implies \text{conflict } S S'$   
 $\langle \text{proof} \rangle$

**lemma** *trancpl-conflict-iff[iff]*:

*full1 conflict*  $S S' \longleftrightarrow \text{conflict } S S'$   
 $\langle \text{proof} \rangle$

**inductive** *cdcl<sub>W</sub>-cp* :: *'st*  $\Rightarrow$  *'st*  $\Rightarrow$  *bool* **where**  
*conflict*'[*intro*]: *conflict*  $S S' \implies \text{cdcl}_W\text{-cp } S S' \mid$   
*propagate*': *propagate*  $S S' \implies \text{cdcl}_W\text{-cp } S S'$

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-rtrancpl-cdcl<sub>W</sub>*:

*cdcl<sub>W</sub>-cp*<sup>\*\*</sup>  $S T \implies \text{cdcl}_W^{\text{**}} S T$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-state-eq-compatible*:

**assumes**

*cdcl<sub>W</sub>-cp*  $S T$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$

**shows** *cdcl<sub>W</sub>-cp*  $S' T'$

$\langle \text{proof} \rangle$

**lemma** *trancpl-cdcl<sub>W</sub>-cp-state-eq-compatible*:

**assumes**

*cdcl<sub>W</sub>-cp*<sup>++</sup>  $S T$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$

**shows** *cdcl<sub>W</sub>-cp*<sup>++</sup>  $S' T'$

$\langle \text{proof} \rangle$

**lemma** *option-full-cdcl<sub>W</sub>-cp*:

*conflicting S ≠ None ⇒ full cdcl<sub>W</sub>-cp S S*  
 ⟨proof⟩

**lemma** *skip-unique*:

*skip S T ⇒ skip S T' ⇒ T ∼ T'*  
 ⟨proof⟩

**lemma** *resolve-unique*:

*resolve S T ⇒ resolve S T' ⇒ T ∼ T'*  
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-cp-no-more-clauses*:

**assumes** *cdcl<sub>W</sub>-cp S S'*  
**shows** *clauses S = clauses S'*  
 ⟨proof⟩

**lemma** *trancpl-cdcl<sub>W</sub>-cp-no-more-clauses*:

**assumes** *cdcl<sub>W</sub>-cp<sup>++</sup> S S'*  
**shows** *clauses S = clauses S'*  
 ⟨proof⟩

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-no-more-clauses*:

**assumes** *cdcl<sub>W</sub>-cp<sup>\*\*</sup> S S'*  
**shows** *clauses S = clauses S'*  
 ⟨proof⟩

**lemma** *no-conflict-after-conflict*:

*conflict S T ⇒ ¬conflict T U*  
 ⟨proof⟩

**lemma** *no-propagate-after-conflict*:

*conflict S T ⇒ ¬propagate T U*  
 ⟨proof⟩

**lemma** *trancpl-cdcl<sub>W</sub>-cp-propagate-with-conflict-or-not*:

**assumes** *cdcl<sub>W</sub>-cp<sup>++</sup> S U*  
**shows** *(propagate<sup>++</sup> S U ∧ conflicting U = None)*  
*∨ (∃ T D. propagate<sup>\*\*</sup> S T ∧ conflict T U ∧ conflicting U = Some D)*  
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-cp-conflicting-not-empty[simp]*: *conflicting S = Some D ⇒ ¬cdcl<sub>W</sub>-cp S S'*

⟨proof⟩

**lemma** *no-step-cdcl<sub>W</sub>-cp-no-conflict-no-propagate*:

**assumes** *no-step cdcl<sub>W</sub>-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<sub>W</sub>-o S S'* and re-apply conflict and propagate *cdcl<sub>W</sub>-cp<sup>↓</sup> S' S''*

**inductive** *cdcl<sub>W</sub>-stgy :: 'st ⇒ 'st ⇒ bool for S :: 'st where*

*conflict': full1 cdcl<sub>W</sub>-cp S S' ⇒ cdcl<sub>W</sub>-stgy S S' |*

*other': cdcl<sub>W</sub>-o S S' ⇒ no-step cdcl<sub>W</sub>-cp S ⇒ full cdcl<sub>W</sub>-cp S' S'' ⇒ cdcl<sub>W</sub>-stgy S S''*



## Invariants

These are the same invariants as before, but lifted

**lemma** *cdcl<sub>W</sub>-cp-learned-clause-inv:*

**assumes** *cdcl<sub>W</sub>-cp S S'*

**shows** *learned-clss S = learned-clss S'*

*<proof>*

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-learned-clause-inv:*

**assumes** *cdcl<sub>W</sub>-cp\*\* S S'*

**shows** *learned-clss S = learned-clss S'*

*<proof>*

**lemma** *trancp-cdcl<sub>W</sub>-cp-learned-clause-inv:*

**assumes** *cdcl<sub>W</sub>-cp<sup>++</sup> S S'*

**shows** *learned-clss S = learned-clss S'*

*<proof>*

**lemma** *cdcl<sub>W</sub>-cp-backtrack-lvl:*

**assumes** *cdcl<sub>W</sub>-cp S S'*

**shows** *backtrack-lvl S = backtrack-lvl S'*

*<proof>*

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-backtrack-lvl:*

**assumes** *cdcl<sub>W</sub>-cp\*\* S S'*

**shows** *backtrack-lvl S = backtrack-lvl S'*

*<proof>*

**lemma** *cdcl<sub>W</sub>-cp-consistent-inv:*

**assumes** *cdcl<sub>W</sub>-cp S S' and cdcl<sub>W</sub>-M-level-inv S*

**shows** *cdcl<sub>W</sub>-M-level-inv S'*

*<proof>*

**lemma** *full1-cdcl<sub>W</sub>-cp-consistent-inv:*

**assumes** *full1 cdcl<sub>W</sub>-cp S S' and cdcl<sub>W</sub>-M-level-inv S*

**shows** *cdcl<sub>W</sub>-M-level-inv S'*

*<proof>*

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-consistent-inv:*

**assumes** *rtrancp cdcl<sub>W</sub>-cp S S' and cdcl<sub>W</sub>-M-level-inv S*

**shows** *cdcl<sub>W</sub>-M-level-inv S'*

*<proof>*

**lemma** *cdcl<sub>W</sub>-stgy-consistent-inv:*

**assumes** *cdcl<sub>W</sub>-stgy S S' and cdcl<sub>W</sub>-M-level-inv S*

**shows** *cdcl<sub>W</sub>-M-level-inv S'*

*<proof>*

**lemma** *rtrancp-cdcl<sub>W</sub>-stgy-consistent-inv:*

**assumes** *cdcl<sub>W</sub>-stgy\*\* S S' and cdcl<sub>W</sub>-M-level-inv S*

**shows** *cdcl<sub>W</sub>-M-level-inv S'*

*<proof>*

**lemma** *cdcl<sub>W</sub>-cp-no-more-init-clss:*

**assumes** *cdcl<sub>W</sub>-cp S S'*

**shows**  $\text{init-clss } S = \text{init-clss } S'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{trancpl-cdcl}_W\text{-cp-no-more-init-clss}$ :

**assumes**  $\text{cdcl}_W\text{-cp}^{++} S S'$   
**shows**  $\text{init-clss } S = \text{init-clss } S'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-stgy-no-more-init-clss}$ :

**assumes**  $\text{cdcl}_W\text{-stgy } S S'$  **and**  $\text{cdcl}_W\text{-M-level-inv } S$   
**shows**  $\text{init-clss } S = \text{init-clss } S'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtrancpl-cdcl}_W\text{-stgy-no-more-init-clss}$ :

**assumes**  $\text{cdcl}_W\text{-stgy}^{**} S S'$  **and**  $\text{cdcl}_W\text{-M-level-inv } S$   
**shows**  $\text{init-clss } S = \text{init-clss } S'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-cp-dropWhile-trail}'$ :

**assumes**  $\text{cdcl}_W\text{-cp } S S'$   
**obtains**  $M$  **where**  $\text{trail } S' = M @ \text{trail } S$  **and**  $(\forall l \in \text{set } M. \neg \text{is-decided } l)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtrancpl-cdcl}_W\text{-cp-dropWhile-trail}'$ :

**assumes**  $\text{cdcl}_W\text{-cp}^{**} S S'$   
**obtains**  $M :: ('v, 'v \text{ clause}) \text{ ann-lits}$  **where**  
 $\text{trail } S' = M @ \text{trail } S$  **and**  $\forall l \in \text{set } M. \neg \text{is-decided } l$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-cp-dropWhile-trail}$ :

**assumes**  $\text{cdcl}_W\text{-cp } S S'$   
**shows**  $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-decided } l)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rtrancpl-cdcl}_W\text{-cp-dropWhile-trail}$ :

**assumes**  $\text{cdcl}_W\text{-cp}^{**} S S'$   
**shows**  $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-decided } l)$   
 $\langle \text{proof} \rangle$

This theorem can be seen as a termination theorem for  $\text{cdcl}_W\text{-cp}$ .

**lemma**  $\text{length-model-le-vars}$ :

**assumes**  
 $\text{no-strange-atm } S$  **and**  
 $\text{no-d: no-dup } (\text{trail } S)$  **and**  
 $\text{finite } (\text{atms-of-mm } (\text{init-clss } S))$   
**shows**  $\text{length } (\text{trail } S) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } S))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-cp-decreasing-measure}$ :

**assumes**  
 $\text{cdcl}_W$ :  $\text{cdcl}_W\text{-cp } S T$  **and**  
 $M\text{-lev}$ :  $\text{cdcl}_W\text{-M-level-inv } S$  **and**  
 $\text{alien}$ :  $\text{no-strange-atm } S$   
**shows**  $(\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S)$   
 $+ (\text{if conflicting } S = \text{None then } 1 \text{ else } 0)) S$   
 $> (\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$

+ (if conflicting  $S = \text{None}$  then 1 else 0))  $T$   
 <proof>

**lemma**  $cdcl_W\text{-cp-wf}$ : wf {(b, a). ( $cdcl_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a$ )  $\wedge cdcl_W\text{-cp } a \ b$ }  
 <proof>

**lemma**  $rtrancp\text{-}cdcl_W\text{-all-struct-inv-cdcl}_W\text{-cp-iff-rtrancp-cdcl}_W\text{-cp}$ :

**assumes**

*lev*:  $cdcl_W\text{-M-level-inv } S$  **and**

*alien*:  $\text{no-strange-atm } S$

**shows**  $(\lambda a \ b. (cdcl_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge cdcl_W\text{-cp } a \ b)^{**} S \ T$

$\longleftrightarrow cdcl_W\text{-cp}^{**} S \ T$

(**is** ?I  $S \ T \longleftrightarrow ?C \ S \ T$ )

<proof>

**lemma**  $cdcl_W\text{-cp-normalized-element}$ :

**assumes**

*lev*:  $cdcl_W\text{-M-level-inv } S$  **and**

$\text{no-strange-atm } S$

**obtains**  $T$  **where** full  $cdcl_W\text{-cp } S \ T$

<proof>

**lemma**  $\text{always-exists-full-cdcl}_W\text{-cp-step}$ :

**assumes**  $\text{no-strange-atm } S$

**shows**  $\exists S''. \text{full } cdcl_W\text{-cp } S \ S''$

<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**  $\text{no-clause-is-false} :: 'st \Rightarrow \text{bool}$  **where**

$\text{no-clause-is-false} \equiv$

$\lambda S. (\text{conflicting } S = \text{None} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{\text{as}} C\text{Not } D))$

**abbreviation**  $\text{conflict-is-false-with-level} :: 'st \Rightarrow \text{bool}$  **where**

$\text{conflict-is-false-with-level } S \equiv \forall D. \text{conflicting } S = \text{Some } D \longrightarrow D \neq \{\#\}$

$\longrightarrow (\exists L \in \# D. \text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S)$

**lemma**  $\text{not-conflict-not-any-negated-init-clss}$ :

**assumes**  $\forall S'. \neg \text{conflict } S \ S'$

**shows**  $\text{no-clause-is-false } S$

<proof>

**lemma**  $\text{full-cdcl}_W\text{-cp-not-any-negated-init-clss}$ :

**assumes** full  $cdcl_W\text{-cp } S \ S'$

**shows**  $\text{no-clause-is-false } S'$

<proof>

**lemma**  $\text{full1-cdcl}_W\text{-cp-not-any-negated-init-clss}$ :

**assumes** full1  $cdcl_W\text{-cp } S \ S'$

**shows**  $\text{no-clause-is-false } S'$

<proof>

**lemma** *cdcl<sub>W</sub>-stgy-not-non-negated-init-clss:*

**assumes** *cdcl<sub>W</sub>-stgy*  $S S'$   
**shows** *no-clause-is-false*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>W</sub>-stgy-not-non-negated-init-clss:*

**assumes** *cdcl<sub>W</sub>-stgy<sup>\*\*</sup>*  $S S'$  **and** *no-clause-is-false*  $S$   
**shows** *no-clause-is-false*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-conflict-ex-lit-of-max-level:*

**assumes**  
*cdcl<sub>W</sub>-cp*  $S S'$  **and**  
*no-clause-is-false*  $S$  **and**  
*cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows** *conflict-is-false-with-level*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *no-chained-conflict:*

**assumes** *conflict*  $S S'$  **and** *conflict*  $S' S''$   
**shows** *False*  
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-propa-or-propa-conf:*

**assumes** *cdcl<sub>W</sub>-cp<sup>\*\*</sup>*  $S U$   
**shows** *propagate<sup>\*\*</sup>*  $S U \vee (\exists T. \text{propagate<sup>**}</sup> } S T \wedge \text{conflict } T U)$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-cdcl<sub>W</sub>-co-conflict-ex-lit-of-max-level:*

**assumes** *full: full cdcl<sub>W</sub>-cp*  $S U$   
**and** *cls-f: no-clause-is-false*  $S$   
**and** *conflict-is-false-with-level*  $S$   
**and** *lev: cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows** *conflict-is-false-with-level*  $U$   
 $\langle \text{proof} \rangle$

## Literal of highest level in decided literals

**definition** *mark-is-false-with-level*  $:: 'st \Rightarrow \text{bool}$  **where**

*mark-is-false-with-level*  $S' \equiv$

$\forall D M1 M2 L. M1 @ \text{Propagated } L D \# M2 = \text{trail } S' \longrightarrow D - \{\#L\} \neq \{\#\}$   
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{count-decided } M1)$

**definition** *no-more-propagation-to-do*  $:: 'st \Rightarrow \text{bool}$  **where**

*no-more-propagation-to-do*  $S \equiv$

$\forall D M M' L. D + \{\#L\} \in \# \text{ clauses } S \longrightarrow \text{trail } S = M' @ M \longrightarrow M \models_{\text{as}} \text{CNot } D$   
 $\longrightarrow \text{undefined-lit } M L \longrightarrow \text{count-decided } M < \text{backtrack-lvl } S$   
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S) L = \text{count-decided } M)$

**lemma** *propagate-no-more-propagation-to-do:*

**assumes** *propagate: propagate*  $S S'$   
**and**  $H: \text{no-more-propagation-to-do } S$   
**and** *lev-inv: cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows** *no-more-propagation-to-do*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *conflict-no-more-propagation-to-do*:

**assumes**

*conflict*: *conflict*  $S$   $S'$  **and**

*H*: *no-more-propagation-to-do*  $S$  **and**

*M*: *cdcl<sub>W</sub>-M-level-inv*  $S$

**shows** *no-more-propagation-to-do*  $S'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-no-more-propagation-to-do*:

**assumes**

*conflict*: *cdcl<sub>W</sub>-cp*  $S$   $S'$  **and**

*H*: *no-more-propagation-to-do*  $S$  **and**

*M*: *cdcl<sub>W</sub>-M-level-inv*  $S$

**shows** *no-more-propagation-to-do*  $S'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-then-exists-cdcl<sub>W</sub>-stgy-step*:

**assumes**

*o*: *cdcl<sub>W</sub>-o*  $S$   $S'$  **and**

*alien*: *no-strange-atm*  $S$  **and**

*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$

**shows**  $\exists S'. \text{cdcl}_W\text{-stgy } S S'$

$\langle \text{proof} \rangle$

**lemma** *backtrack-no-decomp*:

**assumes**

*S*: *conflicting*  $S = \text{Some } E$  **and**

*LE*:  $L \in \# E$  **and**

*L*: *get-level* (*trail*  $S$ )  $L = \text{backtrack-lvl } S$  **and**

*D*: *get-maximum-level* (*trail*  $S$ ) (*remove1-mset*  $L E$ )  $< \text{backtrack-lvl } S$  **and**

*bt*: *backtrack-lvl*  $S = \text{get-maximum-level } (\text{trail } S) E$  **and**

*M-L*: *cdcl<sub>W</sub>-M-level-inv*  $S$

**shows**  $\exists S'. \text{cdcl}_W\text{-o } S S'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-final-state-conclusive*:

**assumes**

*termi*:  $\forall S'. \neg \text{cdcl}_W\text{-stgy } S S'$  **and**

*decomp*: *all-decomposition-implies-m* (*init-clss*  $S$ ) (*get-all-ann-decomposition* (*trail*  $S$ )) **and**

*learned*: *cdcl<sub>W</sub>-learned-clause*  $S$  **and**

*level-inv*: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**

*alien*: *no-strange-atm*  $S$  **and**

*no-dup*: *distinct-cdcl<sub>W</sub>-state*  $S$  **and**

*cnfl*: *cdcl<sub>W</sub>-conflicting*  $S$  **and**

*cnfl-k*: *conflict-is-false-with-level*  $S$

**shows** (*conflicting*  $S = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$ )

$\vee (\text{conflicting } S = \text{None} \wedge \text{trail } S \models_{\text{as set-mset}} (\text{init-clss } S))$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-tranclp-cdcl<sub>W</sub>*:

*cdcl<sub>W</sub>-cp*  $S S' \implies \text{cdcl}_W^{++} S S'$

$\langle \text{proof} \rangle$

**lemma** *tranclp-cdcl<sub>W</sub>-cp-tranclp-cdcl<sub>W</sub>*:

*cdcl<sub>W</sub>-cp*<sup>++</sup>  $S S' \implies \text{cdcl}_W^{++} S S'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-tranclp-cdcl<sub>W</sub>:*  
 $cdcl_W\text{-stgy } S \ S' \implies cdcl_W^{++} \ S \ S'$   
 ⟨proof⟩

**lemma** *tranclp-cdcl<sub>W</sub>-stgy-tranclp-cdcl<sub>W</sub>:*  
 $cdcl_W\text{-stgy}^{++} \ S \ S' \implies cdcl_W^{++} \ S \ S'$   
 ⟨proof⟩

**lemma** *rtranclp-cdcl<sub>W</sub>-stgy-rtranclp-cdcl<sub>W</sub>:*  
 $cdcl_W\text{-stgy}^{**} \ S \ S' \implies cdcl_W^{**} \ S \ S'$   
 ⟨proof⟩

**lemma** *not-empty-get-maximum-level-exists-lit:*  
**assumes**  $n: D \neq \{\#\}$   
**and** *max:*  $get\text{-maximum-level } M \ D = n$   
**shows**  $\exists L \in \#D. get\text{-level } M \ L = n$   
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-o-conflict-is-false-with-level-inv:*  
**assumes**  
 $cdcl_W\text{-o } S \ S'$  **and**  
*lev:*  $cdcl_W\text{-M-level-inv } S$  **and**  
*cnfl-inv:*  $conflict\text{-is-false-with-level } S$  **and**  
*n-d:*  $distinct\text{-cdcl}_W\text{-state } S$  **and**  
*conflicting:*  $cdcl_W\text{-conflicting } S$   
**shows**  $conflict\text{-is-false-with-level } S'$   
 ⟨proof⟩

## Strong completeness

**lemma** *cdcl<sub>W</sub>-cp-propagate-cnfl:*  
**assumes**  $cdcl_W\text{-cp } S \ T$   
**shows**  $propagate^{**} \ S \ T \vee (\exists S'. propagate^{**} \ S \ S' \wedge conflict \ S' \ T)$   
 ⟨proof⟩

**lemma** *rtranclp-cdcl<sub>W</sub>-cp-propagate-cnfl:*  
**assumes**  $cdcl_W\text{-cp}^{**} \ S \ T$   
**shows**  $propagate^{**} \ S \ T \vee (\exists S'. propagate^{**} \ S \ S' \wedge conflict \ S' \ T)$   
 ⟨proof⟩

**lemma** *propagate-high-levelE:*  
**assumes**  $propagate \ S \ T$   
**obtains**  $M' \ N' \ U \ k \ L \ C$  **where**  
 $state \ S = (M', N', U, k, None)$  **and**  
 $state \ T = (Propagated \ L \ (C + \{\#L\# \}) \ \# \ M', N', U, k, None)$  **and**  
 $C + \{\#L\# \} \in \# \ local.clauses \ S$  **and**  
 $M' \models_{as} CNot \ C$  **and**  
 $undefined\text{-lit } (trail \ S) \ L$   
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-cp-propagate-completeness:*  
**assumes**  $MN: set \ M \models_s set\text{-mset } N$  **and**  
*cons:*  $consistent\text{-interp } (set \ M)$  **and**  
*tot:*  $total\text{-over-m } (set \ M) \ (set\text{-mset } N)$  **and**  
*lits-of-l*  $(trail \ S) \subseteq set \ M$  **and**

*init-clss*  $S = N$  **and**  
*propagate\*\**  $S S'$  **and**  
*learned-clss*  $S = \{\#\}$   
**shows**  $\text{length}(\text{trail } S) \leq \text{length}(\text{trail } S') \wedge \text{lits-of-l}(\text{trail } S') \subseteq \text{set } M$   
 ⟨proof⟩

**lemma**

**assumes** *propagate\*\**  $S X$   
**shows**  
*rtrancplp-propagate-init-clss*: *init-clss*  $X = \text{init-clss } S$  **and**  
*rtrancplp-propagate-learned-clss*: *learned-clss*  $X = \text{learned-clss } S$   
 ⟨proof⟩

**lemma** *completeness-is-a-full1-propagation*:

**fixes**  $S :: 'st$  **and**  $M :: 'v$  *literal list*  
**assumes**  $MN$ :  $\text{set } M \models_s \text{set-mset } N$   
**and** *cons*: *consistent-interp* ( $\text{set } M$ )  
**and** *tot*: *total-over-m* ( $\text{set } M$ ) ( $\text{set-mset } N$ )  
**and** *alien*: *no-strange-atm*  $S$   
**and** *learned*: *learned-clss*  $S = \{\#\}$   
**and** *clsS[simp]*: *init-clss*  $S = N$   
**and** *lits*:  $\text{lits-of-l}(\text{trail } S) \subseteq \text{set } M$   
**shows**  $\exists S'. \text{propagate** } S S' \wedge \text{full } \text{cdcl}_W\text{-cp } S S'$   
 ⟨proof⟩

See also *rtrancplp-cdcl<sub>W</sub>-cp-dropWhile-trail*

**lemma** *rtrancplp-propagate-is-trail-append*:

*propagate\*\**  $S T \implies \exists c. \text{trail } T = c @ \text{trail } S$   
 ⟨proof⟩

**lemma** *rtrancplp-propagate-is-update-trail*:

*propagate\*\**  $S T \implies \text{cdcl}_W\text{-M-level-inv } S \implies$   
 $\text{init-clss } S = \text{init-clss } T \wedge \text{learned-clss } S = \text{learned-clss } T \wedge \text{backtrack-lvl } S = \text{backtrack-lvl } T$   
 $\wedge \text{conflicting } S = \text{conflicting } T$   
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-stgy-strong-completeness-n*:

**assumes**  
 $MN$ :  $\text{set } M \models_s \text{set-mset } N$  **and**  
*cons*: *consistent-interp* ( $\text{set } M$ ) **and**  
*tot*: *total-over-m* ( $\text{set } M$ ) ( $\text{set-mset } N$ ) **and**  
*atm-incl*:  $\text{atm-of } '(\text{set } M) \subseteq \text{atms-of-mm } N$  **and**  
*distM*: *distinct*  $M$  **and**  
*length*:  $n \leq \text{length } M$   
**shows**  
 $\exists M' k S. \text{length } M' \geq n \wedge$   
 $\text{lits-of-l } M' \subseteq \text{set } M \wedge$   
 $\text{no-dup } M' \wedge$   
 $\text{state } S = (M', N, \{\#\}, k, \text{None}) \wedge$   
 $\text{cdcl}_W\text{-stgy** } (\text{init-state } N) S$   
 ⟨proof⟩

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

**lemma** *cdcl<sub>W</sub>-stgy-strong-completeness*:

**assumes**  
 $MN$ :  $\text{set } M \models_s \text{set-mset } N$  **and**

*cons*: *consistent-interp* (set  $M$ ) **and**  
*tot*: *total-over-m* (set  $M$ ) (set-mset  $N$ ) **and**  
*atm-incl*: *atm-of* ' (set  $M$ )  $\subseteq$  *atms-of-mm*  $N$  **and**  
*distM*: *distinct*  $M$

**shows**

$\exists M' k S.$

*lits-of-l*  $M' = \text{set } M \wedge$

*state*  $S = (M', N, \{\#\}, k, \text{None}) \wedge$

*cdcl<sub>W</sub>-stgy\*\** (*init-state*  $N$ )  $S \wedge$

*final-cdcl<sub>W</sub>-state*  $S$

$\langle \text{proof} \rangle$

## 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-conflict* ( $S :: 'st$ )  $\equiv$

$(\forall M K M' D. M' @ \text{Decided } K \# M = \text{trail } S \longrightarrow D \in \# \text{ clauses } S$   
 $\longrightarrow \neg M \models_{as} \text{CNot } D)$

**lemma** *no-smaller-conflict-init-sate[simp]*:

*no-smaller-conflict* (*init-state*  $N$ )  $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-o-no-smaller-conflict-inv*:

**fixes**  $S S' :: 'st$

**assumes**

*cdcl<sub>W</sub>-o*  $S S'$  **and**

*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$  **and**

*max-lev*: *conflict-is-false-with-level*  $S$  **and**

*smaller*: *no-smaller-conflict*  $S$  **and**

*no-f*: *no-clause-is-false*  $S$

**shows** *no-smaller-conflict*  $S'$

$\langle \text{proof} \rangle$

**lemma** *conflict-no-smaller-conflict-inv*:

**assumes** *conflict*  $S S'$

**and** *no-smaller-conflict*  $S$

**shows** *no-smaller-conflict*  $S'$

$\langle \text{proof} \rangle$

**lemma** *propagate-no-smaller-conflict-inv*:

**assumes** *propagate*: *propagate*  $S S'$

**and** *n-l*: *no-smaller-conflict*  $S$

**shows** *no-smaller-conflict*  $S'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-no-smaller-conflict-inv*:

**assumes** *propagate*: *cdcl<sub>W</sub>-cp*  $S S'$

**and** *n-l*: *no-smaller-conflict*  $S$

**shows** *no-smaller-conflict*  $S'$

$\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-no-smaller-conflict-inv*:

**assumes** *propagate*: *cdcl<sub>W</sub>-cp\*\**  $S S'$

**and** *n-l*: *no-smaller-conflict*  $S$



**shows** *no-smaller-conflict*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *trancp-cdcl<sub>W</sub>-cp-no-smaller-conflict-inv*:  
**assumes** *propagate*:  $\text{cdcl}_W\text{-cp}^{++} S S'$   
**and** *n-l*: *no-smaller-conflict*  $S$   
**shows** *no-smaller-conflict*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *full-cdcl<sub>W</sub>-cp-no-smaller-conflict-inv*:  
**assumes** *full*  $\text{cdcl}_W\text{-cp} S S'$   
**and** *n-l*: *no-smaller-conflict*  $S$   
**shows** *no-smaller-conflict*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *full1-cdcl<sub>W</sub>-cp-no-smaller-conflict-inv*:  
**assumes** *full1*  $\text{cdcl}_W\text{-cp} S S'$   
**and** *n-l*: *no-smaller-conflict*  $S$   
**shows** *no-smaller-conflict*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-no-smaller-conflict-inv*:  
**assumes**  $\text{cdcl}_W\text{-stgy} S S'$   
**and** *n-l*: *no-smaller-conflict*  $S$   
**and** *conflict-is-false-with-level*  $S$   
**and** *cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows** *no-smaller-conflict*  $S'$   
 $\langle \text{proof} \rangle$

**lemma** *is-conflicting-exists-conflict*:  
**assumes**  $\neg(\forall D \in \# \text{init-clss } S' + \text{learned-clss } S'. \neg \text{trail } S' \models_{\text{as}} \text{CNot } D)$   
**and** *conflicting*  $S' = \text{None}$   
**shows**  $\exists S''. \text{conflict } S' S''$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-o-conflict-is-no-clause-is-false*:  
**fixes**  $S S' :: 'st$   
**assumes**  
*cdcl<sub>W</sub>-o*  $S S'$  **and**  
*lev*: *cdcl<sub>W</sub>-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-conflict*  $S$   
**shows** *no-clause-is-false*  $S'$   
 $\vee (\text{conflicting } S' = \text{None}$   
 $\longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$   
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{backtrack-lvl } S'))$   
 $\langle \text{proof} \rangle$

**lemma** *full1-cdcl<sub>W</sub>-cp-exists-conflict-decompose*:  
**assumes**  
*conflict*:  $\exists D \in \# \text{clauses } S. \text{trail } S \models_{\text{as}} \text{CNot } D$  **and**  
*full*: *full*  $\text{cdcl}_W\text{-cp} S U$  **and**  
*no-conflict*: *conflicting*  $S = \text{None}$  **and**  
*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows**  $\exists T. \text{propagate}^{**} S T \wedge \text{conflict } T U$

*<proof>*

**lemma** *full1-cdcl<sub>W</sub>-cp-exists-conflict-full1-decompose:*

**assumes**

*conf1*:  $\exists D \in \# \text{clauses } S. \text{trail } S \models_{as} CNot \ D$  **and**

*full*: *full cdcl<sub>W</sub>-cp* *S U* **and**

*no-conf1*: *conflicting S = None* **and**

*lev*: *cdcl<sub>W</sub>-M-level-inv S*

**shows**  $\exists T D. \text{propagate}^{**} \ S \ T \wedge \text{conflict } T \ U$

$\wedge \text{trail } T \models_{as} CNot \ D \wedge \text{conflicting } U = \text{Some } D \wedge D \in \# \text{clauses } S$

*<proof>*

**lemma** *cdcl<sub>W</sub>-stgy-no-smaller-conf1:*

**assumes**

*cdcl<sub>W</sub>-stgy S S'* **and**

*n-l*: *no-smaller-conf1 S* **and**

*conflict-is-false-with-level S* **and**

*cdcl<sub>W</sub>-M-level-inv S* **and**

*no-clause-is-false S* **and**

*distinct-cdcl<sub>W</sub>-state S* **and**

*cdcl<sub>W</sub>-conflicting S*

**shows** *no-smaller-conf1 S'*

*<proof>*

**lemma** *cdcl<sub>W</sub>-stgy-ex-lit-of-max-level:*

**assumes**

*cdcl<sub>W</sub>-stgy S S'* **and**

*n-l*: *no-smaller-conf1 S* **and**

*conflict-is-false-with-level S* **and**

*cdcl<sub>W</sub>-M-level-inv S* **and**

*no-clause-is-false S* **and**

*distinct-cdcl<sub>W</sub>-state S* **and**

*cdcl<sub>W</sub>-conflicting S*

**shows** *conflict-is-false-with-level S'*

*<proof>*

**lemma** *rtranc1p-cdcl<sub>W</sub>-stgy-no-smaller-conf1-inv:*

**assumes**

*cdcl<sub>W</sub>-stgy<sup>\*\*</sup> S S'* **and**

*n-l*: *no-smaller-conf1 S* **and**

*cls-false*: *conflict-is-false-with-level S* **and**

*lev*: *cdcl<sub>W</sub>-M-level-inv S* **and**

*no-f*: *no-clause-is-false S* **and**

*dist*: *distinct-cdcl<sub>W</sub>-state S* **and**

*conflicting*: *cdcl<sub>W</sub>-conflicting S* **and**

*decomp*: *all-decomposition-implies-m (init-clss S) (get-all-ann-decomposition (trail S))* **and**

*learned*: *cdcl<sub>W</sub>-learned-clause S* **and**

*alien*: *no-strange-atm S*

**shows** *no-smaller-conf1 S'  $\wedge$  conflict-is-false-with-level S'*

*<proof>*

## Final States are Conclusive

**lemma** *full-cdcl<sub>W</sub>-stgy-final-state-conclusive-non-false:*

**fixes** *S' :: 'st*

**assumes** *full*: *full cdcl<sub>W</sub>-stgy (init-state N) S'*

**and** *no-d*: *distinct-mset-mset*  $N$   
**and** *no-empty*:  $\forall D \in \#N. D \neq \{\#\}$   
**shows** (*conflicting*  $S' = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S'))$ )  
 $\vee (\text{conflicting } S' = \text{None} \wedge \text{trail } S' \models_{\text{asm}} \text{init-clss } S')$   
 $\langle \text{proof} \rangle$

**lemma** *conflict-is-full1-cdcl<sub>W</sub>-cp*:  
**assumes** *cp*: *conflict*  $S S'$   
**shows** *full1 cdcl<sub>W</sub>-cp*  $S S'$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-fst-empty-conflicting-false*:  
**assumes**  
 $\text{cdcl}_W\text{-cp } S S'$  **and**  
 $\text{trail } S = []$  **and**  
 $\text{conflicting } S \neq \text{None}$   
**shows** *False*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-o-fst-empty-conflicting-false*:  
**assumes** *cdcl<sub>W</sub>-o*  $S S'$   
**and**  $\text{trail } S = []$   
**and**  $\text{conflicting } S \neq \text{None}$   
**shows** *False*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-fst-empty-conflicting-false*:  
**assumes** *cdcl<sub>W</sub>-stgy*  $S S'$   
**and**  $\text{trail } S = []$   
**and**  $\text{conflicting } S \neq \text{None}$   
**shows** *False*  
 $\langle \text{proof} \rangle$

**thm** *cdcl<sub>W</sub>-cp.induct[split-format(complete)]*

**lemma** *cdcl<sub>W</sub>-cp-conflicting-is-false*:  
 $\text{cdcl}_W\text{-cp } S S' \implies \text{conflicting } S = \text{Some } \{\#\} \implies \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-conflicting-is-false*:  
 $\text{cdcl}_W\text{-cp}^{++} S S' \implies \text{conflicting } S = \text{Some } \{\#\} \implies \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-o-conflicting-is-false*:  
 $\text{cdcl}_W\text{-o } S S' \implies \text{conflicting } S = \text{Some } \{\#\} \implies \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-conflicting-is-false*:  
 $\text{cdcl}_W\text{-stgy } S S' \implies \text{conflicting } S = \text{Some } \{\#\} \implies \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-stgy-conflicting-is-false*:  
 $\text{cdcl}_W\text{-stgy}^* S S' \implies \text{conflicting } S = \text{Some } \{\#\} \implies S' = S$   
 $\langle \text{proof} \rangle$

**lemma** *full-cdcl<sub>W</sub>-init-clss-with-false-normal-form*:

**assumes**  
 $\forall m \in \text{set } M. \neg \text{is-decided } m$  **and**  
 $E = \text{Some } D$  **and**  
 $\text{state } S = (M, N, U, 0, E)$   
 $\text{full } \text{cdcl}_W\text{-stgy } S \ S'$  **and**  
 $\text{all-decomposition-implies-}m \ (\text{init-clss } S) \ (\text{get-all-ann-decomposition } (\text{trail } S))$   
 $\text{cdcl}_W\text{-learned-clause } S$   
 $\text{cdcl}_W\text{-}M\text{-level-inv } S$   
 $\text{no-strange-atm } S$   
 $\text{distinct-cdcl}_W\text{-state } S$   
 $\text{cdcl}_W\text{-conflicting } S$   
**shows**  $\exists M''. \text{state } S' = (M'', N, U, 0, \text{Some } \{\#\})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{full-cdcl}_W\text{-stgy-final-state-conclusive-is-one-false}$ :  
**fixes**  $S' :: 'st$   
**assumes**  $\text{full: full } \text{cdcl}_W\text{-stgy } (\text{init-state } N) \ S'$   
**and**  $\text{no-d: distinct-mset-mset } N$   
**and**  $\text{empty: } \{\#\} \in \# \ N$   
**shows**  $\text{conflicting } S' = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S'))$   
 $\langle \text{proof} \rangle$

theorem 2.9.9 page 83 of Weidenbach's book

**lemma**  $\text{full-cdcl}_W\text{-stgy-final-state-conclusive}$ :  
**fixes**  $S' :: 'st$   
**assumes**  $\text{full: full } \text{cdcl}_W\text{-stgy } (\text{init-state } N) \ S'$  **and**  $\text{no-d: distinct-mset-mset } N$   
**shows**  $(\text{conflicting } S' = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S')))$   
 $\vee (\text{conflicting } S' = \text{None} \wedge \text{trail } S' \models_{\text{asm}} \text{init-clss } S')$   
 $\langle \text{proof} \rangle$

theorem 2.9.9 page 83 of Weidenbach's book

**lemma**  $\text{full-cdcl}_W\text{-stgy-final-state-conclusive-from-init-state}$ :  
**fixes**  $S' :: 'st$   
**assumes**  $\text{full: full } \text{cdcl}_W\text{-stgy } (\text{init-state } N) \ S'$   
**and**  $\text{no-d: distinct-mset-mset } N$   
**shows**  $(\text{conflicting } S' = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } N))$   
 $\vee (\text{conflicting } S' = \text{None} \wedge \text{trail } S' \models_{\text{asm}} N \wedge \text{satisfiable } (\text{set-mset } N))$   
 $\langle \text{proof} \rangle$

**end**

**end**

**theory**  $\text{CDCL-}W\text{-Termination}$

**imports**  $\text{CDCL-}W$

**begin**

**context**  $\text{conflict-driven-clause-learning}_W$

**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**  $\text{cdcl}_W\text{-all-struct-inv}$  **where**

$cdcl_W\text{-all-struct-inv } S \longleftrightarrow$   
 $no\text{-strange-atm } S \wedge$   
 $cdcl_W\text{-M-level-inv } S \wedge$   
 $(\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s) \wedge$   
 $distinct\text{-}cdcl_W\text{-state } S \wedge$   
 $cdcl_W\text{-conflicting } S \wedge$   
 $all\text{-decomposition-implies-m } (init\text{-clss } S) (get\text{-all-ann-decomposition } (trail\ S)) \wedge$   
 $cdcl_W\text{-learned-clause } S$

**lemma**  $cdcl_W\text{-all-struct-inv-inv}$ :  
**assumes**  $cdcl_W\ S\ S'$  **and**  $cdcl_W\text{-all-struct-inv } S$   
**shows**  $cdcl_W\text{-all-struct-inv } S'$   
 $\langle proof \rangle$

**lemma**  $rtrancp\text{-}cdcl_W\text{-all-struct-inv-inv}$ :  
**assumes**  $cdcl_W^{**}\ S\ S'$  **and**  $cdcl_W\text{-all-struct-inv } S$   
**shows**  $cdcl_W\text{-all-struct-inv } S'$   
 $\langle proof \rangle$

**lemma**  $cdcl_W\text{-stgy-}cdcl_W\text{-all-struct-inv}$ :  
 $cdcl_W\text{-stgy } S\ T \implies cdcl_W\text{-all-struct-inv } S \implies cdcl_W\text{-all-struct-inv } T$   
 $\langle proof \rangle$

**lemma**  $rtrancp\text{-}cdcl_W\text{-stgy-}cdcl_W\text{-all-struct-inv}$ :  
 $cdcl_W\text{-stgy}^{**}\ S\ T \implies cdcl_W\text{-all-struct-inv } S \implies cdcl_W\text{-all-struct-inv } T$   
 $\langle proof \rangle$

## No Relearning of a clause

**lemma**  $cdcl_W\text{-o-new-clause-learned-is-backtrack-step}$ :  
**assumes**  $learned: D \in \# \text{ learned-clss } T$  **and**  
 $new: D \notin \# \text{ learned-clss } S$  **and**  
 $cdcl_W: cdcl_W\text{-o } S\ T$  **and**  
 $lev: cdcl_W\text{-M-level-inv } S$   
**shows**  $backtrack\ S\ T \wedge conflicting\ S = \text{Some } D$   
 $\langle proof \rangle$

**lemma**  $cdcl_W\text{-cp-new-clause-learned-has-backtrack-step}$ :  
**assumes**  $learned: D \in \# \text{ learned-clss } T$  **and**  
 $new: D \notin \# \text{ learned-clss } S$  **and**  
 $cdcl_W: cdcl_W\text{-stgy } S\ T$  **and**  
 $lev: cdcl_W\text{-M-level-inv } S$   
**shows**  $\exists S'. backtrack\ S\ S' \wedge cdcl_W\text{-stgy}^{**}\ S'\ T \wedge conflicting\ S = \text{Some } D$   
 $\langle proof \rangle$

**lemma**  $rtrancp\text{-}cdcl_W\text{-cp-new-clause-learned-has-backtrack-step}$ :  
**assumes**  $learned: D \in \# \text{ learned-clss } T$  **and**  
 $new: D \notin \# \text{ learned-clss } S$  **and**  
 $cdcl_W: cdcl_W\text{-stgy}^{**}\ S\ T$  **and**  
 $lev: cdcl_W\text{-M-level-inv } S$   
**shows**  $\exists S' S''. cdcl_W\text{-stgy}^{**}\ S\ S' \wedge backtrack\ S'\ S'' \wedge conflicting\ S' = \text{Some } D \wedge$   
 $cdcl_W\text{-stgy}^{**}\ S''\ T$   
 $\langle proof \rangle$

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

**shows**  $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$   
 $\langle proof \rangle$

**lemma** *conflict-no-more-Decided-lit:*

**assumes** *conflict*  $S\ S'$   
**shows**  $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$   
 $\langle proof \rangle$

**lemma** *cdcl<sub>W</sub>-cp-no-more-Decided-lit:*

**assumes** *cdcl<sub>W</sub>-cp*  $S\ S'$   
**shows**  $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$   
 $\langle proof \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-no-more-Decided-lit:*

**assumes** *cdcl<sub>W</sub>-cp\*\**  $S\ S'$   
**shows**  $Decided\ K \in set\ (trail\ S) \longleftrightarrow Decided\ K \in set\ (trail\ S')$   
 $\langle proof \rangle$

**lemma** *cdcl<sub>W</sub>-o-no-more-Decided-lit:*

**assumes** *cdcl<sub>W</sub>-o*  $S\ S'$  **and** *lev: cdcl<sub>W</sub>-M-level-inv*  $S$  **and**  $\neg decide\ S\ S'$   
**shows**  $Decided\ K \in set\ (trail\ S') \longrightarrow Decided\ K \in set\ (trail\ S)$   
 $\langle proof \rangle$

**lemma** *cdcl<sub>W</sub>-new-decided-at-beginning-is-decide:*

**assumes** *cdcl<sub>W</sub>-stgy*  $S\ S'$  **and**  
*lev: cdcl<sub>W</sub>-M-level-inv*  $S$  **and**  
 $trail\ S' = M' @ Decided\ L \# M$  **and**  
 $trail\ S = M$   
**shows**  $\exists T. decide\ S\ T \wedge no-step\ cdcl_W-cp\ S$   
 $\langle proof \rangle$

**lemma** *cdcl<sub>W</sub>-o-is-decide:*

**assumes** *cdcl<sub>W</sub>-o*  $S\ T$  **and** *lev: cdcl<sub>W</sub>-M-level-inv*  $S$   
 $trail\ T = drop\ (length\ M_0)\ M' @ Decided\ L \# H @ M$  **and**  
 $\neg (\exists M'. trail\ S = M' @ Decided\ L \# H @ M)$   
**shows**  $decide\ S\ T$   
 $\langle proof \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-new-decided-at-beginning-is-decide:*

**assumes** *cdcl<sub>W</sub>-stgy\*\**  $R\ U$  **and**  
 $trail\ U = M' @ Decided\ L \# H @ M$  **and**  
 $trail\ R = M$  **and**  
*cdcl<sub>W</sub>-M-level-inv*  $R$   
**shows**  
 $\exists S\ T\ T'. cdcl_W-stgy^{**}\ R\ S \wedge decide\ S\ T \wedge cdcl_W-stgy^{**}\ T\ U \wedge cdcl_W-stgy^{**}\ S\ U \wedge$   
 $no-step\ cdcl_W-cp\ S \wedge trail\ T = Decided\ L \# H @ M \wedge trail\ S = H @ M \wedge cdcl_W-stgy\ S\ T' \wedge$   
 $cdcl_W-stgy^{**}\ T'\ U$   
 $\langle proof \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-new-decided-at-beginning-is-decide':*

**assumes** *cdcl<sub>W</sub>-stgy\*\**  $R\ U$  **and**  
 $trail\ U = M' @ Decided\ L \# H @ M$  **and**  
 $trail\ R = M$  **and**  
*cdcl<sub>W</sub>-M-level-inv*  $R$   
**shows**  $\exists y\ y'. cdcl_W-stgy^{**}\ R\ y \wedge cdcl_W-stgy\ y\ y' \wedge \neg (\exists c. trail\ y = c @ Decided\ L \# H @ M)$   
 $\wedge (\lambda a\ b. cdcl_W-stgy\ a\ b \wedge (\exists c. trail\ a = c @ Decided\ L \# H @ M))^{**}\ y'\ U$

$\langle \text{proof} \rangle$

**lemma** *beginning-not-decided-invert:*

**assumes**  $A: M @ A = M' @ \text{Decided } K \# H$  **and**

$nm: \forall m \in \text{set } M. \neg \text{is-decided } m$

**shows**  $\exists M. A = M @ \text{Decided } K \# H$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-trail-has-new-decided-is-decide-step:*

**assumes**  $\text{cdcl}_W\text{-stgy } S \ T$

$\neg (\exists c. \text{trail } S = c @ \text{Decided } L \# H @ M)$  **and**

$(\lambda a b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Decided } L \# H @ M))^{**} \ T \ U$  **and**

$\exists M'. \text{trail } U = M' @ \text{Decided } L \# H @ M$  **and**

$\text{lev: cdcl}_W\text{-M-level-inv } S$

**shows**  $\exists S'. \text{decide } S \ S' \wedge \text{full cdcl}_W\text{-cp } S' \ T \wedge \text{no-step cdcl}_W\text{-cp } S$

$\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-stgy-with-trail-end-has-trail-end:*

**assumes**  $(\lambda a b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Decided } L \# H @ M))^{**} \ T \ U$  **and**

$\exists M'. \text{trail } U = M' @ \text{Decided } L \# H @ M$

**shows**  $\exists M'. \text{trail } T = M' @ \text{Decided } L \# H @ M$

$\langle \text{proof} \rangle$

**lemma** *remove1-mset-eq-remove1-mset-same:*

$\text{remove1-mset } L \ D = \text{remove1-mset } L' \ D \implies L \in \# \ D \implies L = L'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-o-cannot-learn:*

**assumes**

$\text{cdcl}_W\text{-o } y \ z$  **and**

$\text{lev: cdcl}_W\text{-M-level-inv } y$  **and**

$M: \text{trail } y = c @ \text{Decided } Kh \# H$  **and**

$DL: D \notin \# \text{learned-clss } y$  **and**

$LD: L \in \# \ D$  **and**

$DH: \text{atms-of } (\text{remove1-mset } L \ D) \subseteq \text{atm-of } ' \text{ lits-of-l } H$  **and**

$LH: \text{atm-of } L \notin \text{atm-of } ' \text{ lits-of-l } H$  **and**

$\text{learned: } \forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{\text{as}} \text{CNot } T$  **and**

$z: \text{trail } z = c' @ \text{Decided } Kh \# H$

**shows**  $D \notin \# \text{learned-clss } z$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-with-trail-end-has-not-been-learned:*

**assumes**

$\text{cdcl}_W\text{-stgy } y \ z$  **and**

$\text{cdcl}_W\text{-M-level-inv } y$  **and**

$\text{trail } y = c @ \text{Decided } Kh \# H$  **and**

$D \notin \# \text{learned-clss } y$  **and**

$LD: L \in \# \ D$  **and**

$DH: \text{atms-of } (\text{remove1-mset } L \ D) \subseteq \text{atm-of } ' \text{ lits-of-l } H$  **and**

$LH: \text{atm-of } L \notin \text{atm-of } ' \text{ lits-of-l } H$  **and**

$\forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{\text{as}} \text{CNot } T$  **and**

$\text{trail } z = c' @ \text{Decided } Kh \# H$

**shows**  $D \notin \# \text{learned-clss } z$

$\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-stgy-with-trail-end-has-not-been-learned:*

**assumes**

( $\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Decided } K \# H @ [])^{**} S z$  and  
 $\text{cdcl}_W\text{-all-struct-inv } S$  and  
 $\text{trail } S = c @ \text{Decided } K \# H$  and  
 $D \notin \# \text{learned-clss } S$  and  
 $LD: L \in \# D$  and  
 $DH: \text{atms-of } (\text{remove1-mset } L D) \subseteq \text{atm-of } \text{'lits-of-l } H$  and  
 $LH: \text{atm-of } L \notin \text{atm-of } \text{'lits-of-l } H$  and  
 $\exists c'. \text{trail } z = c' @ \text{Decided } K \# H$ )

**shows**  $D \notin \# \text{learned-clss } z$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-stgy-new-learned-clause}$ :

**assumes**  $\text{cdcl}_W\text{-stgy } S T$  and

$\text{lev: cdcl}_W\text{-M-level-inv } S$  and

$E \notin \# \text{learned-clss } S$  and

$E \in \# \text{learned-clss } T$

**shows**  $\exists S'. \text{backtrack } S S' \wedge \text{conflicting } S = \text{Some } E \wedge \text{full } \text{cdcl}_W\text{-cp } S' T$

$\langle \text{proof} \rangle$

theorem 2.9.7 page 83 of Weidenbach's book

**lemma**  $\text{cdcl}_W\text{-stgy-no-relearned-clause}$ :

**assumes**

$\text{invR: cdcl}_W\text{-all-struct-inv } R$  and

$\text{st': cdcl}_W\text{-stgy}^{**} R S$  and

$\text{bt: backtrack } S T$  and

$\text{confl: conflicting } S = \text{Some } E$  and

$\text{already-learned: } E \in \# \text{clauses } S$  and

$R: \text{trail } R = []$

**shows**  $\text{False}$

$\langle \text{proof} \rangle$

**lemma**  $\text{rtrancpl-cdcl}_W\text{-stgy-distinct-mset-clauses}$ :

**assumes**

$\text{invR: cdcl}_W\text{-all-struct-inv } R$  and

$\text{st: cdcl}_W\text{-stgy}^{**} R S$  and

$\text{dist: distinct-mset } (\text{clauses } R)$  and

$R: \text{trail } R = []$

**shows**  $\text{distinct-mset } (\text{clauses } S)$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-stgy-distinct-mset-clauses}$ :

**assumes**

$\text{st: cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$  and

$\text{no-duplicate-clause: distinct-mset } N$  and

$\text{no-duplicate-in-clause: distinct-mset-mset } N$

**shows**  $\text{distinct-mset } (\text{clauses } S)$

$\langle \text{proof} \rangle$

## Decrease of a Measure

**fun**  $\text{cdcl}_W\text{-measure}$  **where**

$\text{cdcl}_W\text{-measure } S =$

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



```

    else length (trail S)
  ]

```

**lemma** *length-model-le-vars-all-inv*:  
**assumes** *cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $\text{length } (\text{trail } S) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } S))$   
 $\langle \text{proof} \rangle$   
**end**

**context** *conflict-driven-clause-learning<sub>W</sub>*  
**begin**

**lemma** *learned-clss-less-upper-bound*:  
**fixes**  $S :: 'st$   
**assumes**  
   *distinct-cdcl<sub>W</sub>-state S* **and**  
    $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$   
**shows**  $\text{card}(\text{set-mset } (\text{learned-clss } S)) \leq 3 \wedge \text{card } (\text{atms-of-mm } (\text{learned-clss } S))$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-measure-decreasing*:  
**fixes**  $S :: 'st$   
**assumes**  
   *cdcl<sub>W</sub> S S'* **and**  
   *no-restart*:  
      $\neg(\text{learned-clss } S \subseteq \# \text{ learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = \text{None})$   
   **and**  
   *no-forget*:  $\text{learned-clss } S \subseteq \# \text{ learned-clss } S'$  **and**  
   *no-relearn*:  $\bigwedge S'. \text{backtrack } S S' \implies \forall T. \text{conflicting } S = \text{Some } T \longrightarrow T \notin \# \text{ learned-clss } S$   
   **and**  
   *alien*: *no-strange-atm S* **and**  
   *M-level*: *cdcl<sub>W</sub>-M-level-inv S* **and**  
   *no-taut*:  $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$  **and**  
   *no-dup*: *distinct-cdcl<sub>W</sub>-state S* **and**  
   *confl*: *cdcl<sub>W</sub>-conflicting S*  
**shows**  $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *propagate-measure-decreasing*:  
**fixes**  $S :: 'st$   
**assumes** *propagate S S'* **and** *cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *conflict-measure-decreasing*:  
**fixes**  $S :: 'st$   
**assumes** *conflict S S'* **and** *cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *decide-measure-decreasing*:  
**fixes**  $S :: 'st$   
**assumes** *decide S S'* **and** *cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-measure-decreasing*:  
**fixes**  $S :: 'st$   
**assumes** *cdcl<sub>W</sub>-cp*  $S S'$  **and** *cdcl<sub>W</sub>-all-struct-inv*  $S$   
**shows**  $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *trancpl-cdcl<sub>W</sub>-cp-measure-decreasing*:  
**fixes**  $S :: 'st$   
**assumes** *cdcl<sub>W</sub>-cp<sup>++</sup>*  $S S'$  **and** *cdcl<sub>W</sub>-all-struct-inv*  $S$   
**shows**  $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-stgy-step-decreasing*:  
**fixes**  $R S T :: 'st$   
**assumes** *cdcl<sub>W</sub>-stgy*  $S T$  **and**  
*cdcl<sub>W</sub>-stgy<sup>\*\*</sup>*  $R S$   
*trail*  $R = []$  **and**  
*cdcl<sub>W</sub>-all-struct-inv*  $R$   
**shows**  $(\text{cdcl}_W\text{-measure } T, \text{cdcl}_W\text{-measure } S) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

Roughly corresponds to theorem 2.9.15 page 86 of Weidenbach's book (using a different bound)

**lemma** *trancpl-cdcl<sub>W</sub>-stgy-decreasing*:  
**fixes**  $R S T :: 'st$   
**assumes** *cdcl<sub>W</sub>-stgy<sup>++</sup>*  $R S$   
*trail*  $R = []$  **and**  
*cdcl<sub>W</sub>-all-struct-inv*  $R$   
**shows**  $(\text{cdcl}_W\text{-measure } S, \text{cdcl}_W\text{-measure } R) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *trancpl-cdcl<sub>W</sub>-stgy-S0-decreasing*:  
**fixes**  $R S T :: 'st$   
**assumes**  
*pl*: *cdcl<sub>W</sub>-stgy<sup>++</sup>*  $(\text{init-state } N) S$  **and**  
*no-dup*: *distinct-mset-mset*  $N$   
**shows**  $(\text{cdcl}_W\text{-measure } S, \text{cdcl}_W\text{-measure } (\text{init-state } N)) \in \text{lexn less-than } 3$   
 $\langle \text{proof} \rangle$

**lemma** *wf-trancpl-cdcl<sub>W</sub>-stgy*:  
*wf*  $\{(S :: 'st, \text{init-state } N) \mid$   
 $S N. \text{distinct-mset-mset } N \wedge \text{cdcl}_W\text{-stgy}^{++} (\text{init-state } N) S\}$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-wf-all-inv*:  
*wf*  $\{(S', S). \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-cp } S S'\}$   
**(is** *wf*  $?R)$   
 $\langle \text{proof} \rangle$

**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<sub>W</sub>.conflict* to find the conflict
2. the conflict is analysed by repetitive application of *conflict-driven-clause-learning<sub>W</sub>.resolve* and *conflict-driven-clause-learning<sub>W</sub>.skip*,
3. finally *conflict-driven-clause-learning<sub>W</sub>.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<sub>W</sub>*  
**begin**  
**declare** *cdcl<sub>W</sub>.intros[intro]* *cdcl<sub>W</sub>-bj.intros[intro]* *cdcl<sub>W</sub>-o.intros[intro]*

**lemma** *backtrack-no-cdcl<sub>W</sub>-bj*:  
**assumes** *cdcl*: *cdcl<sub>W</sub>-bj T U* **and** *inv*: *cdcl<sub>W</sub>-M-level-inv V*  
**shows**  $\neg \text{backtrack } V \ T$   
 $\langle \text{proof} \rangle$

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

**inductive** *skip-or-resolve* :: *'st*  $\Rightarrow$  *'st*  $\Rightarrow$  *bool* **where**  
*s-or-r-skip[intro]*: *skip S T*  $\Longrightarrow$  *skip-or-resolve S T* |  
*s-or-r-resolve[intro]*: *resolve S T*  $\Longrightarrow$  *skip-or-resolve S T*

**lemma** *rtrancpl-cdcl<sub>W</sub>-bj-skip-or-resolve-backtrack*:  
**assumes** *cdcl<sub>W</sub>-bj\*\* S U* **and** *inv*: *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *skip-or-resolve\*\* S U*  $\vee (\exists T. \text{skip-or-resolve** } S \ T \wedge \text{backtrack } T \ U)$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancpl-skip-or-resolve-rtrancpl-cdcl<sub>W</sub>*:  
*skip-or-resolve\*\* S T*  $\Longrightarrow$  *cdcl<sub>W</sub>\*\* S T*  
 $\langle \text{proof} \rangle$

**definition** *backjump-l-cond* :: *'v clause*  $\Rightarrow$  *'v clause*  $\Rightarrow$  *'v literal*  $\Rightarrow$  *'st*  $\Rightarrow$  *'st*  $\Rightarrow$  *bool* **where**  
*backjump-l-cond*  $\equiv \lambda C \ C' \ L' \ S \ T. \ \text{True}$

**definition** *inv<sub>NOT</sub>* :: *'st*  $\Rightarrow$  *bool* **where**  
*inv<sub>NOT</sub>*  $\equiv \lambda S. \text{no-dup } (\text{trail } S)$

**declare** *inv<sub>NOT</sub>-def[simp]*  
**end**

**context** *conflict-driven-clause-learning<sub>W</sub>*  
**begin**

## 6.2.2 More lemmas conflict-propagate and backjumping

### Termination

**lemma** *cdcl<sub>W</sub>-cp-normalized-element-all-inv*:  
**assumes** *inv: cdcl<sub>W</sub>-all-struct-inv S*  
**obtains** *T where full cdcl<sub>W</sub>-cp S T*  
 ⟨*proof*⟩  
**thm** *backtrackE*

**lemma** *cdcl<sub>W</sub>-bj-measure*:  
**assumes** *cdcl<sub>W</sub>-bj S T and cdcl<sub>W</sub>-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<sub>W</sub>-bj*:  
*wf {(b,a). cdcl<sub>W</sub>-bj a b ∧ cdcl<sub>W</sub>-M-level-inv a}*  
 ⟨*proof*⟩

**lemma** *cdcl<sub>W</sub>-bj-exists-normal-form*:  
**assumes** *lev: cdcl<sub>W</sub>-M-level-inv S*  
**shows** *∃ T. full cdcl<sub>W</sub>-bj S T*  
 ⟨*proof*⟩

**lemma** *rtrancp-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** *rtrancp-skip-backtrack-backtrack*:  
**assumes**  
*skip\*\* S T and*  
*backtrack T W and*  
*cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *backtrack S W*  
 ⟨*proof*⟩

See also theorem *rtrancp-skip-backtrack-backtrack*

**lemma** *rtrancp-skip-backtrack-backtrack-end*:  
**assumes**  
*skip: skip\*\* S T and*  
*bt: backtrack S W and*  
*inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *backtrack T W*  
 ⟨*proof*⟩

**lemma** *cdcl<sub>W</sub>-bj-decomp-resolve-skip-and-bj*:  
**assumes** *cdcl<sub>W</sub>-bj\*\* S T* **and** *inv: cdcl<sub>W</sub>-M-level-inv S*  
**shows** (*skip-or-resolve\*\* S T*  
 $\vee (\exists U. \text{skip-or-resolve** } S \ U \wedge \text{backtrack } U \ T)$ )  
 $\langle \text{proof} \rangle$

**lemma** *resolve-skip-deterministic*:  
*resolve S T  $\implies$  skip S U  $\implies$  False*  
 $\langle \text{proof} \rangle$

**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'*  
 $\langle \text{proof} \rangle$

**lemma** *backtrack-unique*:  
**assumes**  
*bt-T: backtrack S T* **and**  
*bt-U: backtrack S U* **and**  
*inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *T  $\sim$  U*  
 $\langle \text{proof} \rangle$

**lemma** *if-can-apply-backtrack-no-more-resolve*:  
**assumes**  
*skip: skip\*\* S U* **and**  
*bt: backtrack S T* **and**  
*inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $\neg \text{resolve } U \ V$   
 $\langle \text{proof} \rangle$

**lemma** *if-can-apply-resolve-no-more-backtrack*:  
**assumes**  
*skip: skip\*\* S U* **and**  
*resolve: resolve S T* **and**  
*inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $\neg \text{backtrack } U \ V$   
 $\langle \text{proof} \rangle$

**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<sub>W</sub>-all-struct-inv S*  
**shows** *skip\*\* S U*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-bj-bj-decomp*:  
**assumes** *cdcl<sub>W</sub>-bj\*\* S W* **and** *cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  
 $(\exists T \ U \ V. (\lambda S \ T. \text{skip-or-resolve } S \ T \wedge \text{no-step backtrack } S)^{**} S \ T$   
 $\wedge (\lambda T \ U. \text{resolve } T \ U \wedge \text{no-step backtrack } T) \ T \ U$   
 $\wedge \text{skip** } U \ V \wedge \text{backtrack } V \ W)$   
 $\vee (\exists T \ U. (\lambda S \ T. \text{skip-or-resolve } S \ T \wedge \text{no-step backtrack } S)^{**} S \ T$

$\wedge (\lambda T U. \text{resolve } T U \wedge \text{no-step backtrack } T) T U \wedge \text{skip}^{**} U W$   
 $\vee (\exists T. \text{skip}^{**} S T \wedge \text{backtrack } T W)$   
 $\vee \text{skip}^{**} S W (\text{is } ?RB S W \vee ?R S W \vee ?SB S W \vee ?S S W)$   
 $\langle \text{proof} \rangle$

The case distinction is needed, since  $T \sim V$  does not imply that  $R^{**} T V$ .

**lemma** *cdcl<sub>W</sub>-bj-strongly-confluent*:

**assumes**  
 $\text{cdcl}_W\text{-bj}^{**} S V$  **and**  
 $\text{cdcl}_W\text{-bj}^{**} S T$  **and**  
 $n\text{-s: no-step cdcl}_W\text{-bj } V$  **and**  
 $\text{inv: cdcl}_W\text{-all-struct-inv } S$   
**shows**  $T \sim V \vee \text{cdcl}_W\text{-bj}^{**} T V$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-bj-unique-normal-form*:

**assumes**  
 $ST: \text{cdcl}_W\text{-bj}^{**} S T$  **and**  $SU: \text{cdcl}_W\text{-bj}^{**} S U$  **and**  
 $n\text{-s-U: no-step cdcl}_W\text{-bj } U$  **and**  
 $n\text{-s-T: no-step cdcl}_W\text{-bj } T$  **and**  
 $\text{inv: cdcl}_W\text{-all-struct-inv } S$   
**shows**  $T \sim U$   
 $\langle \text{proof} \rangle$

**lemma** *full-cdcl<sub>W</sub>-bj-unique-normal-form*:

**assumes** *full cdcl<sub>W</sub>-bj*  $S T$  **and** *full cdcl<sub>W</sub>-bj*  $S U$  **and**  
 $\text{inv: cdcl}_W\text{-all-struct-inv } S$   
**shows**  $T \sim U$   
 $\langle \text{proof} \rangle$

### 6.2.3 CDCL with Merging

**inductive** *cdcl<sub>W</sub>-merge-restart* ::  $'st \Rightarrow 'st \Rightarrow \text{bool}$  **where**

$\text{fw-r-propagate: propagate } S S' \Longrightarrow \text{cdcl}_W\text{-merge-restart } S S' \mid$   
 $\text{fw-r-conflict: conflict } S T \Longrightarrow \text{full cdcl}_W\text{-bj } T U \Longrightarrow \text{cdcl}_W\text{-merge-restart } S U \mid$   
 $\text{fw-r-decide: decide } S S' \Longrightarrow \text{cdcl}_W\text{-merge-restart } S S' \mid$   
 $\text{fw-r-rf: cdcl}_W\text{-rf } S S' \Longrightarrow \text{cdcl}_W\text{-merge-restart } S S'$

**lemma** *rtrancpl-cdcl<sub>W</sub>-bj-rtrancpl-cdcl<sub>W</sub>*:

$\text{cdcl}_W\text{-bj}^{**} S T \Longrightarrow \text{cdcl}_W^{**} S T$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-restart-cdcl<sub>W</sub>*:

**assumes** *cdcl<sub>W</sub>-merge-restart*  $S T$   
**shows**  $\text{cdcl}_W^{**} S T$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-restart-conflicting-true-or-no-step*:

**assumes** *cdcl<sub>W</sub>-merge-restart*  $S T$   
**shows**  $\text{conflicting } T = \text{None} \vee \text{no-step cdcl}_W T$   
 $\langle \text{proof} \rangle$

**inductive** *cdcl<sub>W</sub>-merge* ::  $'st \Rightarrow 'st \Rightarrow \text{bool}$  **where**

$\text{fw-propagate: propagate } S S' \Longrightarrow \text{cdcl}_W\text{-merge } S S' \mid$   
 $\text{fw-conflict: conflict } S T \Longrightarrow \text{full cdcl}_W\text{-bj } T U \Longrightarrow \text{cdcl}_W\text{-merge } S U \mid$

*fw-decide*:  $\text{decide } S \ S' \implies \text{cdcl}_W\text{-merge } S \ S'$

*fw-forget*:  $\text{forget } S \ S' \implies \text{cdcl}_W\text{-merge } S \ S'$

**lemma** *cdcl<sub>W</sub>-merge-cdcl<sub>W</sub>-merge-restart*:

$\text{cdcl}_W\text{-merge } S \ T \implies \text{cdcl}_W\text{-merge-restart } S \ T$

*<proof>*

**lemma** *rtrancp-cdcl<sub>W</sub>-merge-trancp-cdcl<sub>W</sub>-merge-restart*:

$\text{cdcl}_W\text{-merge}^{**} \ S \ T \implies \text{cdcl}_W\text{-merge-restart}^{**} \ S \ T$

*<proof>*

**lemma** *cdcl<sub>W</sub>-merge-rtrancp-cdcl<sub>W</sub>*:

$\text{cdcl}_W\text{-merge } S \ T \implies \text{cdcl}_W^{**} \ S \ T$

*<proof>*

**lemma** *rtrancp-cdcl<sub>W</sub>-merge-rtrancp-cdcl<sub>W</sub>*:

$\text{cdcl}_W\text{-merge}^{**} \ S \ T \implies \text{cdcl}_W^{**} \ S \ T$

*<proof>*

**lemmas** *rulesE* =

*skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE*

**lemma** *cdcl<sub>W</sub>-all-struct-inv-trancp-cdcl<sub>W</sub>-merge-trancp-cdcl<sub>W</sub>-merge-cdcl<sub>W</sub>-all-struct-inv*:

**assumes**

*inv*:  $\text{cdcl}_W\text{-all-struct-inv } b$

$\text{cdcl}_W\text{-merge}^{++} \ b \ a$

**shows**  $(\lambda S \ T. \ \text{cdcl}_W\text{-all-struct-inv } S \ \wedge \ \text{cdcl}_W\text{-merge } S \ T)^{++} \ b \ a$

*<proof>*

**lemma** *backtrack-is-full1-cdcl<sub>W</sub>-bj*:

**assumes** *bt*:  $\text{backtrack } S \ T$  **and** *inv*:  $\text{cdcl}_W\text{-M-level-inv } S$

**shows**  $\text{full1 } \text{cdcl}_W\text{-bj } S \ T$

*<proof>*

**lemma** *rtranc-cdcl<sub>W</sub>-conflicting-true-cdcl<sub>W</sub>-merge-restart*:

**assumes**  $\text{cdcl}_W^{**} \ S \ V$  **and** *inv*:  $\text{cdcl}_W\text{-M-level-inv } S$  **and** *conflicting*  $S = \text{None}$

**shows**  $(\text{cdcl}_W\text{-merge-restart}^{**} \ S \ V \ \wedge \ \text{conflicting } V = \text{None})$

$\vee (\exists T \ U. \ \text{cdcl}_W\text{-merge-restart}^{**} \ S \ T \ \wedge \ \text{conflicting } V \neq \text{None} \ \wedge \ \text{conflict } T \ U \ \wedge \ \text{cdcl}_W\text{-bj}^{**} \ U \ V)$

*<proof>*

**lemma** *no-step-cdcl<sub>W</sub>-no-step-cdcl<sub>W</sub>-merge-restart*:  $\text{no-step } \text{cdcl}_W \ S \implies \text{no-step } \text{cdcl}_W\text{-merge-restart } S$

*<proof>*

**lemma** *no-step-cdcl<sub>W</sub>-merge-restart-no-step-cdcl<sub>W</sub>*:

**assumes**

*conflicting*  $S = \text{None}$  **and**

$\text{cdcl}_W\text{-M-level-inv } S$  **and**

$\text{no-step } \text{cdcl}_W\text{-merge-restart } S$

**shows**  $\text{no-step } \text{cdcl}_W \ S$

*<proof>*

**lemma** *cdcl<sub>W</sub>-merge-restart-no-step-cdcl<sub>W</sub>-bj*:

**assumes**

$\text{cdcl}_W\text{-merge-restart } S \ T$

**shows**  $\text{no-step } \text{cdcl}_W\text{-bj } T$

$\langle \text{proof} \rangle$

**lemma** *rtrancp-cdcl<sub>W</sub>-merge-restart-no-step-cdcl<sub>W</sub>-bj*:

**assumes**

*cdcl<sub>W</sub>-merge-restart*\*\* *S T* **and**

*conflicting S = None*

**shows** *no-step cdcl<sub>W</sub>-bj T*

$\langle \text{proof} \rangle$

If *conflicting S*  $\neq$  *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<sub>W</sub>-iff-full-cdcl<sub>W</sub>-merge*:

**assumes** *confl*: *conflicting S = None* **and** *lev*: *cdcl<sub>W</sub>-M-level-inv S*

**shows** *full cdcl<sub>W</sub> S V*  $\longleftrightarrow$  *full cdcl<sub>W</sub>-merge-restart S V*

$\langle \text{proof} \rangle$

**lemma** *init-state-true-full-cdcl<sub>W</sub>-iff-full-cdcl<sub>W</sub>-merge*:

**shows** *full cdcl<sub>W</sub> (init-state N) V*  $\longleftrightarrow$  *full cdcl<sub>W</sub>-merge-restart (init-state N) V*

$\langle \text{proof} \rangle$

## 6.2.4 CDCL with Merge and Strategy

### The intermediate step

**inductive** *cdcl<sub>W</sub>-s'* :: *'st*  $\Rightarrow$  *'st*  $\Rightarrow$  *bool* **where**

*conflict'*: *full1 cdcl<sub>W</sub>-cp S S'*  $\Longrightarrow$  *cdcl<sub>W</sub>-s' S S'* |

*decide'*: *decide S S'*  $\Longrightarrow$  *no-step cdcl<sub>W</sub>-cp S*  $\Longrightarrow$  *full cdcl<sub>W</sub>-cp S' S''*  $\Longrightarrow$  *cdcl<sub>W</sub>-s' S S''* |

*bj'*: *full1 cdcl<sub>W</sub>-bj S S'*  $\Longrightarrow$  *no-step cdcl<sub>W</sub>-cp S*  $\Longrightarrow$  *full cdcl<sub>W</sub>-cp S' S''*  $\Longrightarrow$  *cdcl<sub>W</sub>-s' S S''*

**inductive-cases** *cdcl<sub>W</sub>-s'E*: *cdcl<sub>W</sub>-s' S T*

**lemma** *rtrancp-cdcl<sub>W</sub>-bj-full1-cdclp-cdcl<sub>W</sub>-stgy*:

*cdcl<sub>W</sub>-bj*\*\* *S S'*  $\Longrightarrow$  *full cdcl<sub>W</sub>-cp S' S''*  $\Longrightarrow$  *cdcl<sub>W</sub>-stgy*\*\* *S S''*

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-s'-is-rtrancp-cdcl<sub>W</sub>-stgy*:

*cdcl<sub>W</sub>-s' S T*  $\Longrightarrow$  *cdcl<sub>W</sub>-stgy*\*\* *S T*

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-cdcl<sub>W</sub>-bj-bissimulation*:

**assumes**

*full cdcl<sub>W</sub>-cp T U* **and**

*cdcl<sub>W</sub>-bj*\*\* *T T'* **and**

*cdcl<sub>W</sub>-all-struct-inv T* **and**

*no-step cdcl<sub>W</sub>-bj T'*

**shows** *full cdcl<sub>W</sub>-cp T' U*

$\vee (\exists U' U''. \text{full cdcl}_W\text{-cp } T' U'' \wedge \text{full1 cdcl}_W\text{-bj } U U' \wedge \text{full cdcl}_W\text{-cp } U' U''$

$\wedge \text{cdcl}_W\text{-s}^{/**} U U'')$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-cdcl<sub>W</sub>-bj-bissimulation'*:

**assumes**

*full cdcl<sub>W</sub>-cp T U* **and**

*cdcl<sub>W</sub>-bj*\*\* *T T'* **and**

*cdcl<sub>W</sub>-all-struct-inv T* **and**



*no-step cdcl<sub>W</sub>-bj T'*  
**shows** *full cdcl<sub>W</sub>-cp T' U*  
 $\vee (\exists U'. \text{full1 } \text{cdcl}_W\text{-bj } U \ U' \wedge (\forall U''. \text{full } \text{cdcl}_W\text{-cp } U' \ U'' \longrightarrow \text{full } \text{cdcl}_W\text{-cp } T' \ U''$   
 $\wedge \text{cdcl}_W\text{-s}^{***} \ U \ U''))$   
 <proof>

**lemma** *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-s'-connected:*  
**assumes** *cdcl<sub>W</sub>-stgy S U and cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *cdcl<sub>W</sub>-s' S U*  
 $\vee (\exists U'. \text{full1 } \text{cdcl}_W\text{-bj } U \ U' \wedge (\forall U''. \text{full } \text{cdcl}_W\text{-cp } U' \ U'' \longrightarrow \text{cdcl}_W\text{-s' } S \ U''))$   
 <proof>

**lemma** *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-s'-connected':*  
**assumes** *cdcl<sub>W</sub>-stgy S U and cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *cdcl<sub>W</sub>-s' S U*  
 $\vee (\exists U' \ U''. \text{cdcl}_W\text{-s' } S \ U'' \wedge \text{full1 } \text{cdcl}_W\text{-bj } U \ U' \wedge \text{full } \text{cdcl}_W\text{-cp } U' \ U'')$   
 <proof>

**lemma** *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-s'-no-step:*  
**assumes** *cdcl<sub>W</sub>-stgy S U and cdcl<sub>W</sub>-all-struct-inv S and no-step cdcl<sub>W</sub>-bj U*  
**shows** *cdcl<sub>W</sub>-s' S U*  
 <proof>

**lemma** *rtrancpl-cdcl<sub>W</sub>-stgy-connected-to-rtrancpl-cdcl<sub>W</sub>-s':*  
**assumes** *cdcl<sub>W</sub>-stgy\*\* S U and inv: cdcl<sub>W</sub>-M-level-inv S*  
**shows** *cdcl<sub>W</sub>-s'\*\*\* S U  $\vee (\exists T. \text{cdcl}_W\text{-s'*** } S \ T \wedge \text{cdcl}_W\text{-bj}^{++} \ T \ U \wedge \text{conflicting } U \neq \text{None})$*   
 <proof>

**lemma** *n-step-cdcl<sub>W</sub>-stgy-iff-no-step-cdcl<sub>W</sub>-cl-cdcl<sub>W</sub>-o:*  
**assumes** *inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *no-step cdcl<sub>W</sub>-s' S  $\longleftrightarrow$  no-step cdcl<sub>W</sub>-cp S  $\wedge$  no-step cdcl<sub>W</sub>-o S (is ?S' S  $\longleftrightarrow$  ?C S  $\wedge$  ?O S)*  
 <proof>

**lemma** *cdcl<sub>W</sub>-s'-trancpl-cdcl<sub>W</sub>:*  
 $\text{cdcl}_W\text{-s' } S \ S' \Longrightarrow \text{cdcl}_W^{++} \ S \ S'$   
 <proof>

**lemma** *trancpl-cdcl<sub>W</sub>-s'-trancpl-cdcl<sub>W</sub>:*  
 $\text{cdcl}_W\text{-s'}^{++} \ S \ S' \Longrightarrow \text{cdcl}_W^{++} \ S \ S'$   
 <proof>

**lemma** *rtrancpl-cdcl<sub>W</sub>-s'-rtrancpl-cdcl<sub>W</sub>:*  
 $\text{cdcl}_W\text{-s'}^{**} \ S \ S' \Longrightarrow \text{cdcl}_W^{**} \ S \ S'$   
 <proof>

**lemma** *full-cdcl<sub>W</sub>-stgy-iff-full-cdcl<sub>W</sub>-s':*  
**assumes** *inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *full cdcl<sub>W</sub>-stgy S T  $\longleftrightarrow$  full cdcl<sub>W</sub>-s' S T (is ?S  $\longleftrightarrow$  ?S')*  
 <proof>

**lemma** *conflict-step-cdcl<sub>W</sub>-stgy-step:*  
**assumes**  
*conflict S T*  
*cdcl<sub>W</sub>-all-struct-inv S*  
**shows**  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$   
 <proof>

**lemma** *decide-step-cdcl<sub>W</sub>-stgy-step*:

**assumes**

*decide S T*

*cdcl<sub>W</sub>-all-struct-inv S*

**shows**  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$

*<proof>*

**lemma** *rtranclp-cdcl<sub>W</sub>-cp-conflicting-Some*:

*cdcl<sub>W</sub>-cp\*\* S T  $\implies$  conflicting S = Some D  $\implies$  S = T*

*<proof>*

**inductive** *cdcl<sub>W</sub>-merge-cp* :: *'st  $\Rightarrow$  'st  $\Rightarrow$  bool* **for** *S :: 'st* **where**

*conflict'*: *conflict S T  $\implies$  full cdcl<sub>W</sub>-bj T U  $\implies$  cdcl<sub>W</sub>-merge-cp S U* |

*propagate'*: *propagate<sup>++</sup> S S'  $\implies$  cdcl<sub>W</sub>-merge-cp S S'*

**lemma** *cdcl<sub>W</sub>-merge-restart-cases*[*consumes 1, case-names conflict propagate*]:

**assumes**

*cdcl<sub>W</sub>-merge-cp S U* **and**

$\bigwedge T. \text{conflict } S \ T \implies \text{full } \text{cdcl}_W\text{-bj } T \ U \implies P$  **and**

*propagate<sup>++</sup> S U  $\implies$  P*

**shows** *P*

*<proof>*

**lemma** *cdcl<sub>W</sub>-merge-cp-rtranclp-cdcl<sub>W</sub>-merge*:

*cdcl<sub>W</sub>-merge-cp S T  $\implies$  cdcl<sub>W</sub>-merge<sup>++</sup> S T*

*<proof>*

**lemma** *rtranclp-cdcl<sub>W</sub>-merge-cp-rtranclp-cdcl<sub>W</sub>*:

*cdcl<sub>W</sub>-merge-cp\*\* S T  $\implies$  cdcl<sub>W</sub>\*\* S T*

*<proof>*

**lemma** *full1-cdcl<sub>W</sub>-bj-no-step-cdcl<sub>W</sub>-bj*:

*full1 cdcl<sub>W</sub>-bj S T  $\implies$  no-step cdcl<sub>W</sub>-cp S*

*<proof>*

## Full Transformation

**inductive** *cdcl<sub>W</sub>-s'-without-decide* **where**

*conflict'-without-decide*[*intro*]: *full1 cdcl<sub>W</sub>-cp S S'  $\implies$  cdcl<sub>W</sub>-s'-without-decide S S'* |

*bj'-without-decide*[*intro*]: *full1 cdcl<sub>W</sub>-bj S S'  $\implies$  no-step cdcl<sub>W</sub>-cp S  $\implies$  full cdcl<sub>W</sub>-cp S' S''  $\implies$  cdcl<sub>W</sub>-s'-without-decide S S''*

**lemma** *rtranclp-cdcl<sub>W</sub>-s'-without-decide-rtranclp-cdcl<sub>W</sub>*:

*cdcl<sub>W</sub>-s'-without-decide\*\* S T  $\implies$  cdcl<sub>W</sub>\*\* S T*

*<proof>*

**lemma** *rtranclp-cdcl<sub>W</sub>-s'-without-decide-rtranclp-cdcl<sub>W</sub>-s'*:

*cdcl<sub>W</sub>-s'-without-decide\*\* S T  $\implies$  cdcl<sub>W</sub>-s'<sup>\*\*</sup> S T*

*<proof>*

**lemma** *rtranclp-cdcl<sub>W</sub>-merge-cp-is-rtranclp-cdcl<sub>W</sub>-s'-without-decide*:

**assumes**

*cdcl<sub>W</sub>-merge-cp\*\* S V*

*conflicting S = None*

**shows**

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

**lemma** *rtrancpl-cdcl<sub>W</sub>-s'-without-decide-is-rtrancpl-cdcl<sub>W</sub>-merge-cp:*

**assumes**

*cdcl<sub>W</sub>-s'-without-decide<sup>\*\*</sup> S V* **and**

*confl: conflicting S = None*

**shows**

$(cdcl_W-merge-cp^{**} S V \wedge conflicting\ V = None)$

$\vee (cdcl_W-merge-cp^{**} S V \wedge conflicting\ V \neq None \wedge no-step\ cdcl_W-cp\ V \wedge no-step\ cdcl_W-bj\ V)$

$\vee (\exists T. cdcl_W-merge-cp^{**} S T \wedge conflict\ T\ V)$

$\langle proof \rangle$

**lemma** *no-step-cdcl<sub>W</sub>-s'-no-ste-cdcl<sub>W</sub>-merge-cp:*

**assumes**

*cdcl<sub>W</sub>-all-struct-inv S*

*conflicting S = None*

*no-step cdcl<sub>W</sub>-s' S*

**shows** *no-step cdcl<sub>W</sub>-merge-cp S*

$\langle proof \rangle$

The *no-step decide S* is needed, since *cdcl<sub>W</sub>-merge-cp* is *cdcl<sub>W</sub>-s'* without *decide*.

**lemma** *conflicting-true-no-step-cdcl<sub>W</sub>-merge-cp-no-step-s'-without-decide:*

**assumes**

*confl: conflicting S = None* **and**

*inv: cdcl<sub>W</sub>-M-level-inv S* **and**

*n-s: no-step cdcl<sub>W</sub>-merge-cp S*

**shows** *no-step cdcl<sub>W</sub>-s'-without-decide S*

$\langle proof \rangle$

**lemma** *conflicting-true-no-step-s'-without-decide-no-step-cdcl<sub>W</sub>-merge-cp:*

**assumes**

*inv: cdcl<sub>W</sub>-all-struct-inv S* **and**

*n-s: no-step cdcl<sub>W</sub>-s'-without-decide S*

**shows** *no-step cdcl<sub>W</sub>-merge-cp S*

$\langle proof \rangle$

**lemma** *no-step-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-cp:*

*no-step cdcl<sub>W</sub>-merge-cp S  $\implies$  cdcl<sub>W</sub>-M-level-inv S  $\implies$  no-step cdcl<sub>W</sub>-cp S*

$\langle proof \rangle$

**lemma** *conflicting-not-true-rtrancpl-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-bj:*

**assumes**

*conflicting S = None* **and**

*cdcl<sub>W</sub>-merge-cp<sup>\*\*</sup> S T*

**shows** *no-step cdcl<sub>W</sub>-bj T*

$\langle proof \rangle$

**lemma** *conflicting-true-full-cdcl<sub>W</sub>-merge-cp-iff-full-cdcl<sub>W</sub>-s'-without-decode:*

**assumes**

*confl: conflicting S = None* **and**

*inv: cdcl<sub>W</sub>-all-struct-inv S*

**shows**

*full cdcl<sub>W</sub>-merge-cp S V  $\longleftrightarrow$  full cdcl<sub>W</sub>-s'-without-decode S V (is ?fw  $\longleftrightarrow$  ?s')*

$\langle \text{proof} \rangle$

**lemma** *conflicting-true-full1-cdcl<sub>W</sub>-merge-cp-iff-full1-cdcl<sub>W</sub>-s'-without-decode:*

**assumes**

*conf*: *conflicting*  $S = \text{None}$  **and**

*inv*: *cdcl<sub>W</sub>-all-struct-inv*  $S$

**shows**

*full1 cdcl<sub>W</sub>-merge-cp*  $S V \longleftrightarrow \text{full1 cdcl<sub>W</sub>-s'-without-decide } S V$

$\langle \text{proof} \rangle$

**lemma** *conflicting-true-full1-cdcl<sub>W</sub>-merge-cp-imp-full1-cdcl<sub>W</sub>-s'-without-decode:*

**assumes**

*fw*: *full1 cdcl<sub>W</sub>-merge-cp*  $S V$  **and**

*inv*: *cdcl<sub>W</sub>-all-struct-inv*  $S$

**shows**

*full1 cdcl<sub>W</sub>-s'-without-decide*  $S V$

$\langle \text{proof} \rangle$

**inductive** *cdcl<sub>W</sub>-merge-stgy* **for**  $S :: 'st$  **where**

*fw-s-cp*[*intro*]: *full1 cdcl<sub>W</sub>-merge-cp*  $S T \implies \text{cdcl}_W\text{-merge-stgy } S T \mid$

*fw-s-decide*[*intro*]: *decide*  $S T \implies \text{no-step cdcl}_W\text{-merge-cp } S \implies \text{full cdcl}_W\text{-merge-cp } T U$   
 $\implies \text{cdcl}_W\text{-merge-stgy } S U$

**lemma** *cdcl<sub>W</sub>-merge-stgy-tranclp-cdcl<sub>W</sub>-merge:*

**assumes** *fw*: *cdcl<sub>W</sub>-merge-stgy*  $S T$

**shows** *cdcl<sub>W</sub>-merge*<sup>++</sup>  $S T$

$\langle \text{proof} \rangle$

**lemma** *rtranclp-cdcl<sub>W</sub>-merge-stgy-rtranclp-cdcl<sub>W</sub>-merge:*

**assumes** *fw*: *cdcl<sub>W</sub>-merge-stgy*<sup>\*\*</sup>  $S T$

**shows** *cdcl<sub>W</sub>-merge*<sup>\*\*</sup>  $S T$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-stgy-rtranclp-cdcl<sub>W</sub>:*

*cdcl<sub>W</sub>-merge-stgy*  $S T \implies \text{cdcl}_W^{**} S T$

$\langle \text{proof} \rangle$

**lemma** *rtranclp-cdcl<sub>W</sub>-merge-stgy-rtranclp-cdcl<sub>W</sub>:*

*cdcl<sub>W</sub>-merge-stgy*<sup>\*\*</sup>  $S T \implies \text{cdcl}_W^{**} S T$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-stgy-cases*[*consumes 1, case-names fw-s-cp fw-s-decide*]:

**assumes**

*cdcl<sub>W</sub>-merge-stgy*  $S U$

*full1 cdcl<sub>W</sub>-merge-cp*  $S U \implies P$

$\bigwedge T. \text{decide } S T \implies \text{no-step cdcl}_W\text{-merge-cp } S \implies \text{full cdcl}_W\text{-merge-cp } T U \implies P$

**shows**  $P$

$\langle \text{proof} \rangle$

**inductive** *cdcl<sub>W</sub>-s'-w*  $:: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **where**

*conflict'*: *full1 cdcl<sub>W</sub>-s'-without-decide*  $S S' \implies \text{cdcl}_W\text{-s'-w } S S' \mid$

*decide'*: *decide*  $S S' \implies \text{no-step cdcl}_W\text{-s'-without-decide } S \implies \text{full cdcl}_W\text{-s'-without-decide } S' S''$   
 $\implies \text{cdcl}_W\text{-s'-w } S S''$

**lemma** *cdcl<sub>W</sub>-s'-w-rtranclp-cdcl<sub>W</sub>:*

*cdcl<sub>W</sub>-s'-w*  $S T \implies \text{cdcl}_W^{**} S T$

$\langle proof \rangle$

**lemma**  $rtrancp-cdcl_W-s'-w-rtrancp-cdcl_W$ :

$cdcl_W-s'-w^{**} S T \implies cdcl_W^{**} S T$

$\langle proof \rangle$

**lemma**  $no-step-cdcl_W-cp-no-step-cdcl_W-s'-without-decide$ :

**assumes**  $no-step\ cdcl_W-cp\ S$  **and**  $conflicting\ S = None$  **and**  $inv: cdcl_W-M-level-inv\ S$

**shows**  $no-step\ cdcl_W-s'-without-decide\ S$

$\langle proof \rangle$

**lemma**  $no-step-cdcl_W-cp-no-step-cdcl_W-merge-restart$ :

**assumes**  $no-step\ cdcl_W-cp\ S$  **and**  $conflicting\ S = None$

**shows**  $no-step\ cdcl_W-merge-cp\ S$

$\langle proof \rangle$

**lemma**  $after-cdcl_W-s'-without-decide-no-step-cdcl_W-cp$ :

**assumes**  $cdcl_W-s'-without-decide\ S\ T$

**shows**  $no-step\ cdcl_W-cp\ T$

$\langle proof \rangle$

**lemma**  $no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-cp$ :

$cdcl_W-all-struct-inv\ S \implies no-step\ cdcl_W-s'-without-decide\ S \implies no-step\ cdcl_W-cp\ S$

$\langle proof \rangle$

**lemma**  $after-cdcl_W-s'-w-no-step-cdcl_W-cp$ :

**assumes**  $cdcl_W-s'-w\ S\ T$  **and**  $cdcl_W-all-struct-inv\ S$

**shows**  $no-step\ cdcl_W-cp\ T$

$\langle proof \rangle$

**lemma**  $rtrancp-cdcl_W-s'-w-no-step-cdcl_W-cp-or-eq$ :

**assumes**  $cdcl_W-s'-w^{**} S\ T$  **and**  $cdcl_W-all-struct-inv\ S$

**shows**  $S = T \vee no-step\ cdcl_W-cp\ T$

$\langle proof \rangle$

**lemma**  $rtrancp-cdcl_W-merge-stgy'-no-step-cdcl_W-cp-or-eq$ :

**assumes**  $cdcl_W-merge-stgy^{**} S\ T$  **and**  $inv: cdcl_W-all-struct-inv\ S$

**shows**  $S = T \vee no-step\ cdcl_W-cp\ T$

$\langle proof \rangle$

**lemma**  $no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj$ :

**assumes**  $no-step\ cdcl_W-s'-without-decide\ S$  **and**  $inv: cdcl_W-all-struct-inv\ S$

**shows**  $no-step\ cdcl_W-bj\ S$

$\langle proof \rangle$

**lemma**  $cdcl_W-s'-w-no-step-cdcl_W-bj$ :

**assumes**  $cdcl_W-s'-w\ S\ T$  **and**  $cdcl_W-all-struct-inv\ S$

**shows**  $no-step\ cdcl_W-bj\ T$

$\langle proof \rangle$

**lemma**  $rtrancp-cdcl_W-s'-w-no-step-cdcl_W-bj-or-eq$ :

**assumes**  $cdcl_W-s'-w^{**} S\ T$  **and**  $cdcl_W-all-struct-inv\ S$

**shows**  $S = T \vee no-step\ cdcl_W-bj\ T$

$\langle proof \rangle$

**lemma**  $rtrancp-cdcl_W-s'-no-step-cdcl_W-s'-without-decide-decomp-into-cdcl_W-merge$ :

**assumes**

$cdcl_W\text{-}s'^{**} R V$  **and**  
 $conflicting R = None$  **and**  
 $inv: cdcl_W\text{-}all\text{-}struct\text{-}inv R$   
**shows**  $(cdcl_W\text{-}merge\text{-}stgy^{**} R V \wedge conflicting V = None)$   
 $\vee (cdcl_W\text{-}merge\text{-}stgy^{**} R V \wedge conflicting V \neq None \wedge no\text{-}step\ cdcl_W\text{-}bj V)$   
 $\vee (\exists S T U. cdcl_W\text{-}merge\text{-}stgy^{**} R S \wedge no\text{-}step\ cdcl_W\text{-}merge\text{-}cp S \wedge decide S T$   
 $\wedge cdcl_W\text{-}merge\text{-}cp^{**} T U \wedge conflict U V)$   
 $\vee (\exists S T. cdcl_W\text{-}merge\text{-}stgy^{**} R S \wedge no\text{-}step\ cdcl_W\text{-}merge\text{-}cp S \wedge decide S T$   
 $\wedge cdcl_W\text{-}merge\text{-}cp^{**} T V$   
 $\wedge conflicting V = None)$   
 $\vee (cdcl_W\text{-}merge\text{-}cp^{**} R V \wedge conflicting V = None)$   
 $\vee (\exists U. cdcl_W\text{-}merge\text{-}cp^{**} R U \wedge conflict U V)$   
 $\langle proof \rangle$

**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**  $cdcl_W\text{-}s'^{**} S U$   
 $\langle proof \rangle$

**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**  $cdcl_W\text{-}s'^{**} R V$   
 $\langle proof \rangle$

**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**  $distinct\text{-}mset (clauses S)$   
 $\langle proof \rangle$

**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**  $no\text{-}step\ cdcl_W\text{-}merge\text{-}stgy R$   
 $\langle proof \rangle$   
**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**  $wf\text{-}tranclp\text{-}cdcl_W\text{-}merge: wf \{(T, S). cdcl_W\text{-}all\text{-}struct\text{-}inv S \wedge cdcl_W\text{-}merge^{++} S T\}$

$\langle proof \rangle$

```

lemma wf-cdclW-merge-cp:
  wf{(T, S). cdclW-all-struct-inv S ∧ cdclW-merge-cp S T}
  ⟨proof⟩

lemma wf-cdclW-merge-stgy:
  wf{(T, S). cdclW-all-struct-inv S ∧ cdclW-merge-stgy S T}
  ⟨proof⟩

lemma cdclW-merge-cp-obtain-normal-form:
  assumes inv: cdclW-all-struct-inv R
  obtains S where full cdclW-merge-cp R S
  ⟨proof⟩

lemma no-step-cdclW-merge-stgy-no-step-cdclW-s':
  assumes
    inv: cdclW-all-struct-inv R and
    confl: conflicting R = None and
    n-s: no-step cdclW-merge-stgy R
  shows no-step cdclW-s' R
  ⟨proof⟩

lemma rtrancp-cdclW-merge-cp-no-step-cdclW-bj:
  assumes conflicting R = None and cdclW-merge-cp** R S
  shows no-step cdclW-bj S
  ⟨proof⟩

lemma rtrancp-cdclW-merge-stgy-no-step-cdclW-bj:
  assumes confl: conflicting R = None and cdclW-merge-stgy** R S
  shows no-step cdclW-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*)  $\longleftrightarrow$  *no-dup M*  
 ⟨proof⟩

**lemma** *convert-trail-from-W-true-annots[simp]*:  
*convert-trail-from-W M*  $\models_{as} C \longleftrightarrow M \models_{as} C$   
 ⟨proof⟩

**lemma** *defined-lit-convert-trail-from-W[simp]*:  
*defined-lit* (*convert-trail-from-W S*) *L*  $\longleftrightarrow$  *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*  $\Rightarrow$  ('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*  $\equiv$  *map convert-ann-lit-from-NOT*

**lemma** *undefined-lit-convert-trail-from-NOT[simp]*:  
*undefined-lit* (*convert-trail-from-NOT F*) *L*  $\longleftrightarrow$  *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<sub>NOT</sub>* **where**  
*trail<sub>NOT</sub> S*  $\equiv$  *convert-trail-from-W (fst S)*

**lemma** *undefined-lit-convert-trail-from-W[iff]*:  
*undefined-lit* (*convert-trail-from-W M*) *L*  $\longleftrightarrow$  *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  $state_W \subseteq dpll\text{-}state\text{-}ops$ 
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  clauses
   $\lambda L \ S. \text{cons-trail (convert-ann-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 
   $\langle proof \rangle$ 

sublocale  $state_W \subseteq dpll\text{-}state$ 
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  clauses
   $\lambda L \ S. \text{cons-trail (convert-ann-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 
   $\langle proof \rangle$ 

context  $state_W$ 
begin
declare  $state\text{-}simp_{NOT}[simp \ del]$ 
end

sublocale  $\text{conflict-driven-clause-learning}_W \subseteq cdcl_{NOT}\text{-merge-bj-learn-ops}$ 
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  clauses
   $\lambda L \ S. \text{cons-trail (convert-ann-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 
   $\lambda - . \text{True}$ 
   $\lambda - \ S. \text{conflicting } S = \text{None}$ 
   $\lambda C \ C' \ L' \ S \ T. \text{backjump-l-cond } C \ C' \ L' \ S \ T$ 
   $\wedge \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$ 
   $\langle proof \rangle$ 

thm  $cdcl_{NOT}\text{-merge-bj-learn-proxy.axioms}$ 
sublocale  $\text{conflict-driven-clause-learning}_W \subseteq cdcl_{NOT}\text{-merge-bj-learn-proxy}$ 
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  clauses
   $\lambda L \ S. \text{cons-trail (convert-ann-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 

   $\lambda - . \text{True}$ 
   $\lambda - \ S. \text{conflicting } S = \text{None}$ 
  backjump-l-cond
   $inv_{NOT}$ 
   $\langle proof \rangle$ 

sublocale  $\text{conflict-driven-clause-learning}_W \subseteq cdcl_{NOT}\text{-merge-bj-learn-proxy2}$ 
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  clauses
   $\lambda L \ S. \text{cons-trail (convert-ann-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 

```

$\lambda C S. \text{add-learned-cls } C S$   
 $\lambda C S. \text{remove-cls } C S$   
 $\lambda -. \text{True}$   
 $\lambda S. \text{conflicting } S = \text{None backjump-l-cond inv}_{NOT}$   
 $\langle \text{proof} \rangle$

**sublocale** *conflict-driven-clause-learning<sub>W</sub>*  $\subseteq$  *cdcl<sub>NOT</sub>-merge-bj-learn*  
 $\lambda S. \text{convert-trail-from-} W \text{ (trail } S)$   
*clauses*  
 $\lambda L S. \text{cons-trail (convert-ann-lit-from-NOT } L) S$   
 $\lambda S. \text{tl-trail } S$   
 $\lambda C S. \text{add-learned-cls } C S$   
 $\lambda C S. \text{remove-cls } C S$   
*backjump-l-cond*  
 $\lambda -. \text{True}$   
 $\lambda S. \text{conflicting } S = \text{None inv}_{NOT}$   
 $\langle \text{proof} \rangle$

**context** *conflict-driven-clause-learning<sub>W</sub>*  
**begin**

Notations are lost while proving locale inclusion:

**notation** *state-eq<sub>NOT</sub>* (**infix**  $\sim_{NOT}$  50)

### 6.3.2 Additional Lemmas between NOT and W states

**lemma** *trail<sub>W</sub>-eq-reduce-trail-to<sub>NOT</sub>-eq*:  
 $\text{trail } S = \text{trail } T \implies \text{trail (reduce-trail-to}_{NOT} F S) = \text{trail (reduce-trail-to}_{NOT} F T)$   
 $\langle \text{proof} \rangle$

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-add-learned-cls*:  
 $\text{no-dup (trail } S) \implies$   
 $\text{trail (reduce-trail-to}_{NOT} M (\text{add-learned-cls } D S)) = \text{trail (reduce-trail-to}_{NOT} M S)$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-trail-to<sub>NOT</sub>-reduce-trail-convert*:  
 $\text{reduce-trail-to}_{NOT} C S = \text{reduce-trail-to (convert-trail-from-NOT } C) S$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-trail-to-map[simp]*:  
 $\text{reduce-trail-to (map } f M) S = \text{reduce-trail-to } M S$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-trail-to<sub>NOT</sub>-map[simp]*:  
 $\text{reduce-trail-to}_{NOT} (\text{map } f M) S = \text{reduce-trail-to}_{NOT} M S$   
 $\langle \text{proof} \rangle$

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

### 6.3.3 Inclusion of Weidenbach's CDCL in NOT's CDCL

This lemma shows the inclusion of Weidenbach's CDCL *cdcl<sub>W</sub>-merge* (with merging) in NOT's *cdcl<sub>NOT</sub>-merged-bj-learn*.

**lemma** *cdcl<sub>W</sub>-merge-is-cdcl<sub>NOT</sub>-merged-bj-learn*:

**assumes**

*inv*: *cdcl<sub>W</sub>-all-struct-inv S* **and**

*cdcl<sub>W</sub>*: *cdcl<sub>W</sub>-merge S T*

**shows** *cdcl<sub>NOT</sub>-merged-bj-learn S T*

$\vee$  (*no-step cdcl<sub>W</sub>-merge T*  $\wedge$  *conflicting T*  $\neq$  *None*)

$\langle$ *proof* $\rangle$

**abbreviation** *cdcl<sub>NOT</sub>-restart* **where**

*cdcl<sub>NOT</sub>-restart*  $\equiv$  *restart-ops.cdcl<sub>NOT</sub>-raw-restart cdcl<sub>NOT</sub> restart*

**lemma** *cdcl<sub>W</sub>-merge-restart-is-cdcl<sub>NOT</sub>-merged-bj-learn-restart-no-step*:

**assumes**

*inv*: *cdcl<sub>W</sub>-all-struct-inv S* **and**

*cdcl<sub>W</sub>*: *cdcl<sub>W</sub>-merge-restart S T*

**shows** *cdcl<sub>NOT</sub>-restart\*\* S T*  $\vee$  (*no-step cdcl<sub>W</sub>-merge T*  $\wedge$  *conflicting T*  $\neq$  *None*)

$\langle$ *proof* $\rangle$

**abbreviation**  $\mu_{FW} :: 'st \Rightarrow nat$  **where**

$\mu_{FW} S \equiv$  (*if no-step cdcl<sub>W</sub>-merge S then 0 else 1*  $+$   $\mu_{CDCL}$ -merged (*set-mset (init-cls S)*) *S*)

**lemma** *cdcl<sub>W</sub>-merge- $\mu_{FW}$ -decreasing*:

**assumes**

*inv*: *cdcl<sub>W</sub>-all-struct-inv S* **and**

*fw*: *cdcl<sub>W</sub>-merge S T*

**shows**  $\mu_{FW} T < \mu_{FW} S$

$\langle$ *proof* $\rangle$

**lemma** *wf-cdcl<sub>W</sub>-merge*: *wf*  $\{(T, S). \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T\}$

$\langle$ *proof* $\rangle$

**sublocale** *conflict-driven-clause-learning<sub>W</sub>-termination*

$\langle$ *proof* $\rangle$

### 6.3.4 Correctness of *cdcl<sub>W</sub>-merge-stgy*

**lemma** *full-cdcl<sub>W</sub>-s'-full-cdcl<sub>W</sub>-merge-restart*:

**assumes**

*conflicting R* = *None* **and**

*inv*: *cdcl<sub>W</sub>-all-struct-inv R*

**shows** *full cdcl<sub>W</sub>-s' R V*  $\longleftrightarrow$  *full cdcl<sub>W</sub>-merge-stgy R V* (**is**  $?s' \longleftrightarrow ?fw$ )

$\langle$ *proof* $\rangle$

**lemma** *full-cdcl<sub>W</sub>-stgy-full-cdcl<sub>W</sub>-merge*:

**assumes**

*conflicting R* = *None* **and**

*cdcl<sub>W</sub>-all-struct-inv R*

**shows** *full cdcl<sub>W</sub>-stgy R V*  $\longleftrightarrow$  *full cdcl<sub>W</sub>-merge-stgy R V*

$\langle$ *proof* $\rangle$

**lemma** *full-cdcl<sub>W</sub>-merge-stgy-final-state-conclusive'*:

**fixes**  $S' :: 'st$

```

assumes
  full: full cdclW-merge-stgy (init-state N) S' and
  no-d: distinct-mset-mset N
shows (conflicting S' = Some {#}  $\wedge$  unsatisfiable (set-mset N))
   $\vee$  (conflicting S' = None  $\wedge$  trail S'  $\models_{asm}$  N  $\wedge$  satisfiable (set-mset N))
<proof>
end

end
theory CDCL-W-Restart
imports CDCL-W-Merge
begin

```

### 6.3.5 Adding Restarts

```

locale cdclW-restart =
  conflict-driven-clause-learningW
  — functions for the state:
  — access functions:
  trail init-clss learned-clss backtrack-lvl conflicting
  — changing state:
  cons-trail tl-trail add-learned-clss remove-clss update-backtrack-lvl
  update-conflicting

  — get state:
  init-state
for
  trail :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits and
  init-clss :: 'st  $\Rightarrow$  'v clauses and
  learned-clss :: 'st  $\Rightarrow$  'v clauses and
  backtrack-lvl :: 'st  $\Rightarrow$  nat and
  conflicting :: 'st  $\Rightarrow$  'v clause option and

  cons-trail :: ('v, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-learned-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st +
fixes f :: nat  $\Rightarrow$  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<sub>W</sub>-merge-with-restart **where**

restart-step:

```

  (cdclW-merge-stgy  $\sim$  (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) S T
 $\implies$  card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > f n
 $\implies$  restart T U  $\implies$  cdclW-merge-with-restart (S, n) (U, Suc n) |

```

restart-full: full1 cdcl<sub>W</sub>-merge-stgy S T  $\implies$  cdcl<sub>W</sub>-merge-with-restart (S, n) (T, Suc n)

**lemma** cdcl<sub>W</sub>-merge-with-restart S T  $\implies$  cdcl<sub>W</sub>-merge-restart\*\* (fst S) (fst T)

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-with-restart-rtrancpl-cdcl<sub>W</sub>:*  
*cdcl<sub>W</sub>-merge-with-restart S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> (fst S) (fst T)*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-with-restart-increasing-number:*  
*cdcl<sub>W</sub>-merge-with-restart S T  $\implies$  snd T = 1 + snd S*  
 $\langle \text{proof} \rangle$

**lemma** *full1 cdcl<sub>W</sub>-merge-stgy S T  $\implies$  cdcl<sub>W</sub>-merge-with-restart (S, n) (T, Suc n)*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-all-struct-inv-learned-clss-bound:*  
**assumes** *inv: cdcl<sub>W</sub>-all-struct-inv S*  
**shows** *set-mset (learned-clss S)  $\subseteq$  simple-clss (atms-of-mm (init-clss S))*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-with-restart-init-clss:*  
*cdcl<sub>W</sub>-merge-with-restart S T  $\implies$  cdcl<sub>W</sub>-M-level-inv (fst S)  $\implies$   
init-clss (fst S) = init-clss (fst T)*  
 $\langle \text{proof} \rangle$

**lemma**  
*wf {(T, S). cdcl<sub>W</sub>-all-struct-inv (fst S)  $\wedge$  cdcl<sub>W</sub>-merge-with-restart S T}*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-with-restart-distinct-mset-clauses:*  
**assumes** *invR: cdcl<sub>W</sub>-all-struct-inv (fst R) and*  
*st: cdcl<sub>W</sub>-merge-with-restart R S and*  
*dist: distinct-mset (clauses (fst R)) and*  
*R: trail (fst R) = []*  
**shows** *distinct-mset (clauses (fst S))*  
 $\langle \text{proof} \rangle$

**inductive** *cdcl<sub>W</sub>-with-restart where*

*restart-step:*

*(cdcl<sub>W</sub>-stgy  $\sim$  (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) S T  $\implies$   
card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > f n  $\implies$   
restart T U  $\implies$*

*cdcl<sub>W</sub>-with-restart (S, n) (U, Suc n) |*

*restart-full: full1 cdcl<sub>W</sub>-stgy S T  $\implies$  cdcl<sub>W</sub>-with-restart (S, n) (T, Suc n)*

**lemma** *cdcl<sub>W</sub>-with-restart-rtrancpl-cdcl<sub>W</sub>:*  
*cdcl<sub>W</sub>-with-restart S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> (fst S) (fst T)*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-with-restart-increasing-number:*  
*cdcl<sub>W</sub>-with-restart S T  $\implies$  snd T = 1 + snd S*  
 $\langle \text{proof} \rangle$

**lemma** *full1 cdcl<sub>W</sub>-stgy S T  $\implies$  cdcl<sub>W</sub>-with-restart (S, n) (T, Suc n)*  
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-with-restart-init-clss:*  
*cdcl<sub>W</sub>-with-restart S T  $\implies$  cdcl<sub>W</sub>-M-level-inv (fst S)  $\implies$  init-clss (fst S) = init-clss (fst T)*

$\langle proof \rangle$

**lemma**

*wf*  $\{(T, S). \text{cdcl}_W\text{-all-struct-inv } (fst\ S) \wedge \text{cdcl}_W\text{-with-restart } S\ T\}$

$\langle proof \rangle$

**lemma** *cdcl<sub>W</sub>-with-restart-distinct-mset-clauses:*

**assumes** *invR*: *cdcl<sub>W</sub>-all-struct-inv* (*fst R*) **and**

*st*: *cdcl<sub>W</sub>-with-restart* *R S* **and**

*dist*: *distinct-mset* (*clauses* (*fst R*)) **and**

*R*: *trail* (*fst R*) = []

**shows** *distinct-mset* (*clauses* (*fst S*))

$\langle proof \rangle$

**end**

**locale** *luby-sequence* =

**fixes** *ur* :: *nat*

**assumes** *ur* > 0

**begin**

**lemma** *exists-luby-decomp*:

**fixes** *i* :: *nat*

**shows**  $\exists k :: nat. (2^{\wedge} (k - 1) \leq i \wedge i < 2^{\wedge} k - 1) \vee i = 2^{\wedge} k - 1$

$\langle proof \rangle$

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*  $\Rightarrow$  *nat* **where**

*luby-sequence-core* *i* =

(if  $\exists k. i = 2^{\wedge} k - 1$

then  $2^{\wedge} ((SOME\ k. i = 2^{\wedge} k - 1) - 1)$

else *luby-sequence-core* ( $i - 2^{\wedge} ((SOME\ k. 2^{\wedge} (k-1) \leq i \wedge i < 2^{\wedge} k - 1) - 1) + 1$ ))

$\langle proof \rangle$

**termination**

$\langle proof \rangle$

**function** *natlog2* :: *nat*  $\Rightarrow$  *nat* **where**

*natlog2* *n* = (if *n* = 0 then 0 else 1 + *natlog2* (*n* div 2))

$\langle proof \rangle$

**termination**  $\langle proof \rangle$

**declare** *natlog2.simps*[*simp del*]

**declare** *luby-sequence-core.simps*[*simp del*]

**lemma** *two-pover-n-eq-two-power-n'-eq*:

**assumes** *H*:  $(2::nat)^{\wedge} (k::nat) - 1 = 2^{\wedge} k' - 1$

**shows**  $k' = k$

$\langle proof \rangle$

**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*:  $2^k - \text{Suc } 0 \leq i$  **and**  
   *k'*:  $i < 2^{k'} - \text{Suc } 0$  **and**  
   *k-k'*:  $k > k'$   
**shows** *False*  
 ⟨proof⟩

**lemma** *luby-sequence-core-not-two-power-minus-one*:  
**assumes**  
   *k-i*:  $2^k - 1 \leq i$  **and**  
   *i-k*:  $i < 2^k - 1$   
**shows** *luby-sequence-core i = luby-sequence-core (i - 2^k + 1)*  
 ⟨proof⟩

**lemma** *unbounded-luby-sequence-core: unbounded luby-sequence-core*  
 ⟨proof⟩

**abbreviation** *luby-sequence* :: nat  $\Rightarrow$  nat **where**  
*luby-sequence* n  $\equiv$  *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  $\geq$  1*  
 ⟨proof⟩  
**end**

**locale** *luby-sequence-restart* =  
   *luby-sequence ur* +  
   *conflict-driven-clause-learning<sub>W</sub>*  
   — 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  $\Rightarrow$  ('v, 'v clause) ann-lits **and**  
   *hd-trail* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lit **and**  
   *init-clss* :: 'st  $\Rightarrow$  'v clauses **and**  
   *learned-clss* :: 'st  $\Rightarrow$  'v clauses **and**  
   *backtrack-lvl* :: 'st  $\Rightarrow$  nat **and**  
   *conflicting* :: 'st  $\Rightarrow$  'v clause option **and**

```

    cons-trail :: ('v, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
    tl-trail :: 'st  $\Rightarrow$  'st and
    add-learned-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    remove-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
    update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

    init-state :: 'v clauses  $\Rightarrow$  'st
begin

sublocale cdclW-restart - - - - - luby-sequence
  <proof>

end
end
theory CDCL-W-Incremental
imports CDCL-W-Termination
begin

```

## 6.4 Incremental SAT solving

```

locale stateW-adding-init-clause =
  stateW
  — functions about the state:
  — getter:
  trail init-clss learned-clss backtrack-lvl conflicting
  — setter:
  cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — Some specific states:
  init-state
for
  trail :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits and
  init-clss :: 'st  $\Rightarrow$  'v clauses and
  learned-clss :: 'st  $\Rightarrow$  'v clauses and
  backtrack-lvl :: 'st  $\Rightarrow$  nat and
  conflicting :: 'st  $\Rightarrow$  'v clause option and

  cons-trail :: ('v, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-learned-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st +
fixes
  add-init-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
assumes
  add-init-cls:
    state st = (M, N, U, S')  $\Rightarrow$ 
    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  $\longleftrightarrow$  *conflicting* *S* = None  
⟨*proof*⟩

**end**

**locale** *conflict-driven-clause-learning-with-adding-init-clause<sub>W</sub>* =  
*state<sub>W</sub>-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  $\Rightarrow$  ('v, 'v clause) ann-lits **and**

*hd-trail* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lit **and**

*init-clss* :: 'st  $\Rightarrow$  'v clauses **and**

*learned-clss* :: 'st  $\Rightarrow$  'v clauses **and**

*backtrack-lvl* :: 'st  $\Rightarrow$  nat **and**

*conflicting* :: 'st  $\Rightarrow$  'v clause option **and**

*cons-trail* :: ('v, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*tl-trail* :: 'st  $\Rightarrow$  'st **and**

*add-learned-cls* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*remove-cls* :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*update-backtrack-lvl* :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*update-conflicting* :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

$init-state :: 'v\ clauses \Rightarrow 'st$  **and**  
 $add-init-cls :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$   
**begin**

**sublocale** *conflict-driven-clause-learning<sub>W</sub>*  
 $\langle proof \rangle$

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl<sub>W</sub>-all-struct-inv*

**definition** *cdcl<sub>W</sub>-stgy-invariant* **where**  
 $cdcl_W-stgy-invariant\ S \longleftrightarrow$   
 $conflict-is-false-with-level\ S$   
 $\wedge no-clause-is-false\ S$   
 $\wedge no-smaller-confl\ S$   
 $\wedge no-clause-is-false\ S$

**lemma** *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-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**  
 $cdcl_W-stgy-invariant\ T$   
 $\langle proof \rangle$

**lemma** *rtranchp-cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-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**  
 $cdcl_W-stgy-invariant\ T$   
 $\langle proof \rangle$

**abbreviation** *decr-bt-lvl* **where**  
 $decr-bt-lvl\ S \equiv update-backtrack-lvl\ (backtrack-lvl\ S - 1)\ S$

When we add a new clause, we reduce the trail until we get to the first literal included in C. Then we can mark the conflict.

**fun** *cut-trail-wrt-clause* **where**  
 $cut-trail-wrt-clause\ C\ []\ S = S\ |$   
 $cut-trail-wrt-clause\ C\ (Decided\ L\ \# \ M)\ S =$   
 $(if\ -L \in \# \ C\ then\ S$   
 $\quad else\ cut-trail-wrt-clause\ C\ M\ (decr-bt-lvl\ (tl-trail\ S)))\ |$   
 $cut-trail-wrt-clause\ C\ (Propagated\ L\ - \ \# \ M)\ S =$   
 $(if\ -L \in \# \ C\ then\ S$   
 $\quad else\ cut-trail-wrt-clause\ C\ M\ (tl-trail\ S))$

**definition** *add-new-clause-and-update*  $:: 'v\ clause \Rightarrow 'st \Rightarrow 'st$  **where**  
 $add-new-clause-and-update\ C\ S =$   
 $(if\ trail\ S \models_{as}\ CNot\ C$   
 $\quad then\ update-conflicting\ (Some\ C)\ (add-init-cls\ C$   
 $\quad \quad (cut-trail-wrt-clause\ C\ (trail\ S)\ S))$   
 $\quad else\ add-init-cls\ C\ S)$

**thm** *cut-trail-wrt-clause.induct*

**lemma** *init-clss-cut-trail-wrt-clause[simp]*:

*init-clss* (*cut-trail-wrt-clause* *C M S*) = *init-clss S*  
 ⟨*proof*⟩

**lemma** *learned-clss-cut-trail-wrt-clause[simp]*:

*learned-clss* (*cut-trail-wrt-clause* *C M S*) = *learned-clss S*  
 ⟨*proof*⟩

**lemma** *conflicting-clss-cut-trail-wrt-clause[simp]*:

*conflicting* (*cut-trail-wrt-clause* *C M S*) = *conflicting S*  
 ⟨*proof*⟩

**lemma** *trail-cut-trail-wrt-clause*:

$\exists M. \text{ trail } S = M @ \text{ trail } (\text{cut-trail-wrt-clause } C (\text{trail } S) S)$   
 ⟨*proof*⟩

**lemma** *n-dup-no-dup-trail-cut-trail-wrt-clause[simp]*:

**assumes** *n-d*: *no-dup* (*trail T*)  
**shows** *no-dup* (*trail* (*cut-trail-wrt-clause* *C* (*trail T*) *T*))  
 ⟨*proof*⟩

**lemma** *cut-trail-wrt-clause-backtrack-lvl-length-decided*:

**assumes**  
*backtrack-lvl T* = *count-decided* (*trail T*)  
**shows**  
*backtrack-lvl* (*cut-trail-wrt-clause* *C* (*trail T*) *T*) =  
*count-decided* (*trail* (*cut-trail-wrt-clause* *C* (*trail T*) *T*))  
 ⟨*proof*⟩

**lemma** *cut-trail-wrt-clause-CNot-trail*:

**assumes** *trail T*  $\models_{as}$  *CNot C*  
**shows**  
*trail* ((*cut-trail-wrt-clause* *C* (*trail T*) *T*)))  $\models_{as}$  *CNot C*  
 ⟨*proof*⟩

**lemma** *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*:

$((\forall L \in \#C. \neg L \in \text{ lits-of-l } (\text{trail } T)) \wedge \text{ trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) = [])$   
 $\vee (\neg \text{ lit-of } (\text{hd } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))) \in \# C$   
 $\wedge \text{ length } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \geq 1$   
 ⟨*proof*⟩

We can fully run *cdcl<sub>W</sub>*-s or add a clause. Remark that we use *cdcl<sub>W</sub>*-s to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict *C* if possible.

**inductive** *incremental-cdcl<sub>W</sub>* :: '*st*  $\Rightarrow$  '*st*  $\Rightarrow$  *bool* **for** *S* **where**

*add-conf*:

*trail S*  $\models_{asm}$  *init-clss S*  $\Rightarrow$  *distinct-mset C*  $\Rightarrow$  *conflicting S* = *None*  $\Rightarrow$   
*trail S*  $\models_{as}$  *CNot C*  $\Rightarrow$   
*full cdcl<sub>W</sub>-stgy*  
*(update-conflicting* (*Some C*)  
*(add-init-cls C* (*cut-trail-wrt-clause C* (*trail S*) *S*))) *T*  $\Rightarrow$   
*incremental-cdcl<sub>W</sub> S T* |

*add-no-conf*:

*trail S*  $\models_{asm}$  *init-clss S*  $\Rightarrow$  *distinct-mset C*  $\Rightarrow$  *conflicting S* = *None*  $\Rightarrow$   
 $\neg \text{ trail } S \models_{as} \text{ CNot } C \Rightarrow$   
*full cdcl<sub>W</sub>-stgy* (*add-init-cls C S*) *T*  $\Rightarrow$

*incremental-cdcl<sub>W</sub> S T*

**lemma** *cdcl<sub>W</sub>-all-struct-inv-add-new-clause-and-update-cdcl<sub>W</sub>-all-struct-inv:*

**assumes**

*inv-T: cdcl<sub>W</sub>-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<sub>W</sub>-all-struct-inv (add-new-clause-and-update C T) (is cdcl<sub>W</sub>-all-struct-inv ?T')*  
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-all-struct-inv-add-new-clause-and-update-cdcl<sub>W</sub>-stgy-inv:*

**assumes**

*inv-s: cdcl<sub>W</sub>-stgy-invariant T and*  
*inv: cdcl<sub>W</sub>-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<sub>W</sub>-stgy-invariant (add-new-clause-and-update C T)*  
*(is cdcl<sub>W</sub>-stgy-invariant ?T')*

⟨proof⟩

**lemma** *full-cdcl<sub>W</sub>-stgy-inv-normal-form:*

**assumes**

*full: full cdcl<sub>W</sub>-stgy S T and*  
*inv-s: cdcl<sub>W</sub>-stgy-invariant S and*  
*inv: cdcl<sub>W</sub>-all-struct-inv S*

**shows** *conflicting T = Some {#}  $\wedge$  unsatisfiable (set-mset (init-clss S))*  
 *$\vee$  conflicting T = None  $\wedge$  trail T  $\models_{asm}$  init-clss S  $\wedge$  satisfiable (set-mset (init-clss S))*

⟨proof⟩

**lemma** *incremental-cdcl<sub>W</sub>-inv:*

**assumes**

*inc: incremental-cdcl<sub>W</sub> S T and*  
*inv: cdcl<sub>W</sub>-all-struct-inv S and*  
*s-inv: cdcl<sub>W</sub>-stgy-invariant S*

**shows**

*cdcl<sub>W</sub>-all-struct-inv T and*  
*cdcl<sub>W</sub>-stgy-invariant T*

⟨proof⟩

**lemma** *rtrancpl-incremental-cdcl<sub>W</sub>-inv:*

**assumes**

*inc: incremental-cdcl<sub>W</sub>\*\* S T and*  
*inv: cdcl<sub>W</sub>-all-struct-inv S and*  
*s-inv: cdcl<sub>W</sub>-stgy-invariant S*

**shows**

*cdcl<sub>W</sub>-all-struct-inv T and*  
*cdcl<sub>W</sub>-stgy-invariant T*

⟨proof⟩

**lemma** *incremental-conclusive-state:*

**assumes**

*inc: incremental-cdcl<sub>W</sub> S T and*  
*inv: cdcl<sub>W</sub>-all-struct-inv S and*  
*s-inv: cdcl<sub>W</sub>-stgy-invariant S*

**shows** *conflicting*  $T = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } T))$   
 $\vee \text{conflicting } T = \text{None} \wedge \text{trail } T \models_{\text{asm}} \text{init-clss } T \wedge \text{satisfiable } (\text{set-mset } (\text{init-clss } T))$   
 $\langle \text{proof} \rangle$

**lemma** *trancpl-incremental-correct*:

**assumes**

*inc*: *incremental-cdcl<sub>W</sub><sup>++</sup> S T* **and**

*inv*: *cdcl<sub>W</sub>-all-struct-inv S* **and**

*s-inv*: *cdcl<sub>W</sub>-stgy-invariant S*

**shows** *conflicting*  $T = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } T))$

$\vee \text{conflicting } T = \text{None} \wedge \text{trail } T \models_{\text{asm}} \text{init-clss } T \wedge \text{satisfiable } (\text{set-mset } (\text{init-clss } T))$

$\langle \text{proof} \rangle$

**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 and simply iterate over-and-over on lists.

#### 7.1.1 Common Rules

##### Propagation

The following theorem holds:

**lemma** *lits-of-l-unfold*[iff]:  
 $(\forall c \in \text{set } C. -c \in \text{lits-of-l } Ms) \longleftrightarrow Ms \models_{as} CNot \ (mset \ C)$   
*<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  $\Rightarrow$  ('a, 'b) ann-lits  $\Rightarrow$  'a literal option

**where**

*is-unit-clause* *l* *M* =  
(case *List.filter* ( $\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of-l } M$ ) *l* of  
  *a* # []  $\Rightarrow$  if *M*  $\models_{as} CNot \ (mset \ l - \ \{\#a\# \})$  then *Some a* else *None*  
  | -  $\Rightarrow$  *None*)

**definition** *is-unit-clause-code* :: 'a literal list  $\Rightarrow$  ('a, 'b) ann-lits

$\Rightarrow$  'a literal option **where**

*is-unit-clause-code* *l* *M* =  
(case *List.filter* ( $\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of-l } M$ ) *l* of  
  *a* # []  $\Rightarrow$  if  $(\forall c \in \text{set } (remove1 \ a \ l). -c \in \text{lits-of-l } M)$  then *Some a* else *None*  
  | -  $\Rightarrow$  *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*  $l\ M = \text{Some } a$   
 **shows** *undefined-lit*  $M\ a$   
*<proof>*

**lemma** *is-unit-clause-some-CNot*: *is-unit-clause*  $l\ M = \text{Some } a \implies M \models_{as} \text{CNot } (\text{mset } l - \{\#a\# \})$   
*<proof>*

**lemma** *is-unit-clause-some-in*: *is-unit-clause*  $l\ M = \text{Some } a \implies a \in \text{set } l$   
*<proof>*

**lemma** *is-unit-clause-Nil[simp]*: *is-unit-clause*  $[]\ M = \text{None}$   
*<proof>*

## Unit propagation for all clauses

Finding the first clause to propagate

**fun** *find-first-unit-clause* :: *'a literal list list*  $\Rightarrow$  (*'a, 'b*) *ann-lits*  
  $\Rightarrow$  (*'a literal*  $\times$  *'a literal list*) *option* **where**  
*find-first-unit-clause* ( $a \# l$ )  $M =$   
 (*case is-unit-clause*  $a\ M$  of  
 *None*  $\Rightarrow$  *find-first-unit-clause*  $l\ M$   
 | *Some*  $L \Rightarrow \text{Some } (L, a)$ ) |  
*find-first-unit-clause*  $[]$  - = *None*

**lemma** *find-first-unit-clause-some*:  
 *find-first-unit-clause*  $l\ M = \text{Some } (a, c)$   
  $\implies c \in \text{set } l \wedge M \models_{as} \text{CNot } (\text{mset } c - \{\#a\# \}) \wedge \text{undefined-lit } M\ a \wedge a \in \text{set } c$   
*<proof>*

**lemma** *propagate-is-unit-clause-not-None*:  
 **assumes** *dist*: *distinct*  $c$  **and**  
  $M: M \models_{as} \text{CNot } (\text{mset } c - \{\#a\# \})$  **and**  
 *undef*: *undefined-lit*  $M\ a$  **and**  
 *ac*:  $a \in \text{set } c$   
 **shows** *is-unit-clause*  $c\ M \neq \text{None}$   
*<proof>*

**lemma** *find-first-unit-clause-none*:  
 *distinct*  $c \implies c \in \text{set } l \implies M \models_{as} \text{CNot } (\text{mset } c - \{\#a\# \}) \implies \text{undefined-lit } M\ a \implies a \in \text{set } c$   
  $\implies \text{find-first-unit-clause } l\ M \neq \text{None}$   
*<proof>*

## Decide

**fun** *find-first-unused-var* :: *'a literal list list*  $\Rightarrow$  *'a literal set*  $\Rightarrow$  *'a literal option* **where**  
*find-first-unused-var* ( $a \# l$ )  $M =$   
 (*case List.find* ( $\lambda \text{lit. lit} \notin M \wedge \neg \text{lit} \notin M$ )  $a$  of  
 *None*  $\Rightarrow$  *find-first-unused-var*  $l\ M$   
 | *Some*  $a \Rightarrow \text{Some } a$ ) |  
*find-first-unused-var*  $[]$  - = *None*

**lemma** *find-none[iff]*:



*List.find* ( $\lambda lit. lit \notin M \wedge \neg lit \notin M$ )  $a = None \longleftrightarrow atm\text{-}of\ 'a\ set\ a \subseteq atm\text{-}of\ 'a\ M$   
 <proof>

**lemma** *find-some*: *List.find* ( $\lambda lit. lit \notin M \wedge \neg lit \notin M$ )  $a = Some\ b \implies b \in set\ a \wedge b \notin M \wedge \neg b \notin M$   
 <proof>

**lemma** *find-first-unused-var-None*[iff]:  
*find-first-unused-var*  $l\ M = None \longleftrightarrow (\forall a \in set\ l. atm\text{-}of\ 'a\ set\ a \subseteq atm\text{-}of\ 'a\ M)$   
 <proof>

**lemma** *find-first-unused-var-Some-not-all-incl*:  
**assumes** *find-first-unused-var*  $l\ M = Some\ c$   
**shows**  $\neg(\forall a \in set\ l. atm\text{-}of\ 'a\ set\ a \subseteq atm\text{-}of\ 'a\ M)$   
 <proof>

**lemma** *find-first-unused-var-Some*:  
*find-first-unused-var*  $l\ M = Some\ a \implies (\exists m \in set\ l. a \in set\ m \wedge a \notin M \wedge \neg a \notin M)$   
 <proof>

**lemma** *find-first-unused-var-undefined*:  
*find-first-unused-var*  $l\ (lits\text{-}of\ l\ Ms) = Some\ a \implies undefined\text{-}lit\ Ms\ a$   
 <proof>

## 7.1.2 CDCL specific functions

### Level

**fun** *maximum-level-code*::  $'a\ literal\ list \Rightarrow ('a, 'b)\ ann\text{-}lits \Rightarrow nat$   
**where**  
*maximum-level-code* [] = 0 |  
*maximum-level-code* ( $L\ \# Ls$ )  $M = \max\ (get\text{-}level\ M\ L)\ (maximum\text{-}level\text{-}code\ Ls\ M)$

**lemma** *maximum-level-code-eq-get-maximum-level*[simp]:  
*maximum-level-code*  $D\ M = get\text{-}maximum\text{-}level\ M\ (mset\ D)$   
 <proof>

**lemma** [code]:  
**fixes**  $M :: ('a, 'b)\ ann\text{-}lits$   
**shows**  $get\text{-}maximum\text{-}level\ M\ (mset\ D) = maximum\text{-}level\text{-}code\ D\ M$   
 <proof>

### Backjumping

**fun** *find-level-decomp* **where**  
*find-level-decomp*  $M\ []\ D\ k = None$  |  
*find-level-decomp*  $M\ (L\ \# Ls)\ D\ k =$   
 (case ( $get\text{-}level\ M\ L, maximum\text{-}level\text{-}code\ (D\ @\ Ls)\ M$ ) 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**  $L \in set\ Ls \wedge get\text{-}maximum\text{-}level\ M\ (mset\ (remove1\ L\ (Ls\ @\ D))) = j \wedge get\text{-}level\ M\ L = k$   
 <proof>

**lemma** *find-level-decomp-none*:

**assumes** *find-level-decomp*  $M\ Ls\ E\ k = \text{None}$  **and**  $\text{mset } (L\#D) = \text{mset } (Ls\ @\ E)$   
**shows**  $\neg(L \in \text{set } Ls \wedge \text{get-maximum-level } M\ (\text{mset } D) < k \wedge k = \text{get-level } M\ L)$   
 $\langle \text{proof} \rangle$

**fun** *bt-cut* **where**

*bt-cut*  $i\ (\text{Propagated } -\ -\ \# Ls) = \text{bt-cut } i\ Ls\ |$

*bt-cut*  $i\ (\text{Decided } K\ \# Ls) = (\text{if count-decided } Ls = i \text{ then } \text{Some } (\text{Decided } K\ \# Ls) \text{ else } \text{bt-cut } i\ Ls)\ |$

*bt-cut*  $i\ [] = \text{None}$

**lemma** *bt-cut-some-decomp*:

**assumes** *no-dup*  $M$  **and** *bt-cut*  $i\ M = \text{Some } M'$

**shows**  $\exists K\ M2\ M1. M = M2\ @\ M' \wedge M' = \text{Decided } K\ \# M1 \wedge \text{get-level } M\ K = (i+1)$

$\langle \text{proof} \rangle$

**lemma** *bt-cut-not-none*:

**assumes** *no-dup*  $M$  **and**  $M = M2\ @\ \text{Decided } K\ \# M'$  **and**  $\text{get-level } M\ K = (i+1)$

**shows** *bt-cut*  $i\ M \neq \text{None}$

$\langle \text{proof} \rangle$

**lemma** *get-all-ann-decomposition-ex*:

$\exists N. (\text{Decided } K\ \# M', N) \in \text{set } (\text{get-all-ann-decomposition } (M2\ @\ \text{Decided } K\ \# M'))$

$\langle \text{proof} \rangle$

**lemma** *bt-cut-in-get-all-ann-decomposition*:

**assumes** *no-dup*  $M$  **and** *bt-cut*  $i\ M = \text{Some } M'$

**shows**  $\exists M2. (M', M2) \in \text{set } (\text{get-all-ann-decomposition } M)$

$\langle \text{proof} \rangle$

**fun** *do-backtrack-step* **where**

*do-backtrack-step*  $(M, N, U, k, \text{Some } D) =$

$(\text{case find-level-decomp } M\ D\ []\ k\ \text{of}$

$\text{None} \Rightarrow (M, N, U, k, \text{Some } D)$

$| \text{Some } (L, j) \Rightarrow$

$(\text{case bt-cut } j\ M\ \text{of}$

$\text{Some } (\text{Decided } -\ \# Ls) \Rightarrow (\text{Propagated } L\ D\ \# Ls, N, D\ \# U, j, \text{None})$

$| - \Rightarrow (M, N, U, k, \text{Some } D))$

$)\ |$

*do-backtrack-step*  $S = S$

**end**

**theory** *DPLL-W-Implementation*

**imports** *DPLL-CDCL-W-Implementation* *DPLL-W*  $\sim \sim \text{src/HOL/Library/Code-Target-Numeral}$

**begin**

### 7.1.3 Simple Implementation of DPLL

#### Combining the propagate and decide: a DPLL step

**definition** *DPLL-step* ::  $\text{int dpll}_W\text{-ann-lits} \times \text{int literal list list}$

$\Rightarrow \text{int dpll}_W\text{-ann-lits} \times \text{int literal list list}$  **where**

*DPLL-step* =  $(\lambda(Ms, N).$

$(\text{case find-first-unit-clause } N\ Ms\ \text{of}$

$\text{Some } (L, -) \Rightarrow (\text{Propagated } L\ ()\ \# Ms, N)$

$| - \Rightarrow$

$\text{if } \exists C \in \text{set } N. (\forall c \in \text{set } C. -c \in \text{lits-of-l } Ms)$

$\text{then}$

```

    (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**  $toS \equiv \lambda(Ms::(int, unit) \text{ ann-lits})$   
 $(N:: int \text{ literal list list}). (Ms, mset (map mset N))$   
**abbreviation**  $toS' \equiv \lambda(Ms::(int, unit) \text{ ann-lits},$   
 $N:: int \text{ literal list list}). (Ms, mset (map mset N))$

Proof of correctness of *DPLL-step*

**lemma** *DPLL-step-is-a-dpll<sub>W</sub>-step*:  
**assumes**  $step: (Ms', N') = DPLL\text{-}step (Ms, N)$   
**and**  $neg: (Ms, N) \neq (Ms', N')$   
**shows**  $dpll_W (toS Ms N) (toS Ms' N')$   
 $\langle proof \rangle$

**lemma** *DPLL-step-stuck-final-state*:  
**assumes**  $step: (Ms, N) = DPLL\text{-}step (Ms, N)$   
**shows**  $conclusive\text{-}dpll_W\text{-}state (toS Ms N)$   
 $\langle proof \rangle$

## Adding invariants

**Invariant tested in the function** **function**  $DPLL\text{-}ci :: int \text{ dpll}_W\text{-}ann\text{-}lits \Rightarrow int \text{ literal list list}$   
 $\Rightarrow int \text{ dpll}_W\text{-}ann\text{-}lits \times int \text{ literal list list}$  **where**  
 $DPLL\text{-}ci Ms N =$   
 $(if \neg dpll_W\text{-}all\text{-}inv (Ms, mset (map mset N))$   
 $then (Ms, N)$   
 $else$   
 $let (Ms', N') = DPLL\text{-}step (Ms, N) \text{ in}$   
 $if (Ms', N') = (Ms, N) \text{ then } (Ms, N) \text{ else } DPLL\text{-}ci Ms' N)$   
 $\langle proof \rangle$

**termination**  
 $\langle proof \rangle$

**No invariant tested** **function**  $(domintros) DPLL\text{-}part :: int \text{ dpll}_W\text{-}ann\text{-}lits \Rightarrow int \text{ literal list list} \Rightarrow$   
 $int \text{ dpll}_W\text{-}ann\text{-}lits \times int \text{ literal list list}$  **where**  
 $DPLL\text{-}part Ms N =$   
 $(let (Ms', N') = DPLL\text{-}step (Ms, N) \text{ in}$   
 $if (Ms', N') = (Ms, N) \text{ then } (Ms, N) \text{ else } DPLL\text{-}part Ms' N)$   
 $\langle proof \rangle$

**lemma** *snd-DPLL-step[simp]*:  
 $snd (DPLL\text{-}step (Ms, N)) = N$   
 $\langle proof \rangle$

**lemma** *dpll<sub>W</sub>-all-inv-implicS-2-eq3-and-dom*:  
**assumes** *dpll<sub>W</sub>-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<sub>W</sub>-rtrancp*:  
**assumes** *DPLL-ci Ms N = (Ms', N')*  
**shows** *dpll<sub>W</sub>\*\* (toS Ms N) (toS Ms' N)*  
*<proof>*

**lemma** *dpll<sub>W</sub>-all-inv-dpll<sub>W</sub>-trancp-irrefl*:  
**assumes** *dpll<sub>W</sub>-all-inv* (*Ms*, *N*)  
**and** *dpll<sub>W</sub>\*\* (Ms, N) (Ms, N)*  
**shows** *False*  
*<proof>*

**lemma** *DPLL-ci-final-state*:  
**assumes** *step: DPLL-ci Ms N = (Ms, N)*  
**and** *inv: dpll<sub>W</sub>-all-inv (toS Ms N)*  
**shows** *conclusive-dpll<sub>W</sub>-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<sub>W</sub>-all-inv-final*:  
**fixes** *M Ms':: (int, unit) ann-lits and*  
*N :: int literal list list*  
**assumes** *inv: dpll<sub>W</sub>-all-inv (Ms, mset (map mset N))*  
**and** *MsN: DPLL-part Ms N = (Ms', N)*  
**shows** *conclusive-dpll<sub>W</sub>-state (toS Ms' N) ∧ dpll<sub>W</sub>\*\* (toS Ms N) (toS Ms' N)*  
*<proof>*

## Embedding the invariant into the type

**Defining the type** **typedef** *dpll<sub>W</sub>-state* =  
 {(*M::(int, unit) ann-lits, N::int literal list list*).  
*dpll<sub>W</sub>-all-inv (toS M N)*}  
**morphisms** *rough-state-of state-of*  
*<proof>*

**lemma**

*DPLL-part-dom* ( $\square$ ,  $N$ )  
 $\langle \text{proof} \rangle$

**Some type classes instantiation** *dpll<sub>W</sub>-state* :: *equal*  
**begin**  
**definition** *equal-dpll<sub>W</sub>-state* :: *dpll<sub>W</sub>-state*  $\Rightarrow$  *dpll<sub>W</sub>-state*  $\Rightarrow$  *bool* **where**  
*equal-dpll<sub>W</sub>-state*  $S S' = (\text{rough-state-of } S = \text{rough-state-of } S')$   
**instance**  
 $\langle \text{proof} \rangle$   
**end**

**DPLL definition** *DPLL-step'* :: *dpll<sub>W</sub>-state*  $\Rightarrow$  *dpll<sub>W</sub>-state* **where**  
*DPLL-step'*  $S = \text{state-of } (\text{DPLL-step } (\text{rough-state-of } S))$

**declare** *rough-state-of-inverse*[*simp*]

**lemma** *DPLL-step-dpll<sub>W</sub>-conc-inv*:  
*DPLL-step* (*rough-state-of*  $S$ )  $\in \{(M, N). \text{dpll}_W\text{-all-inv } (\text{toS } M N)\}$   
 $\langle \text{proof} \rangle$

**lemma** *rough-state-of-DPLL-step'-DPLL-step*[*simp*]:  
*rough-state-of* (*DPLL-step'*  $S$ ) = *DPLL-step* (*rough-state-of*  $S$ )  
 $\langle \text{proof} \rangle$

**function** *DPLL-tot*:: *dpll<sub>W</sub>-state*  $\Rightarrow$  *dpll<sub>W</sub>-state* **where**  
*DPLL-tot*  $S =$   
 (*let*  $S' = \text{DPLL-step}' S$  *in*  
   *if*  $S' = S$  *then*  $S$  *else* *DPLL-tot*  $S'$ )  
 $\langle \text{proof} \rangle$   
**termination**  
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
*DPLL-tot*  $S =$   
 (*let*  $S' = \text{DPLL-step}' S$  *in*  
   *if*  $S' = S$  *then*  $S$  *else* *DPLL-tot*  $S'$ )  $\langle \text{proof} \rangle$

**lemma** *DPLL-tot-DPLL-step-DPLL-tot*[*simp*]: *DPLL-tot* (*DPLL-step'*  $S$ ) = *DPLL-tot*  $S$   
 $\langle \text{proof} \rangle$

**lemma** *DOPLL-step'-DPLL-tot*[*simp*]:  
*DPLL-step'* (*DPLL-tot*  $S$ ) = *DPLL-tot*  $S$   
 $\langle \text{proof} \rangle$

**lemma** *DPLL-tot-final-state*:  
**assumes** *DPLL-tot*  $S = S$   
**shows** *conclusive-dpll<sub>W</sub>-state* (*toS'* (*rough-state-of*  $S$ ))  
 $\langle \text{proof} \rangle$

**lemma** *DPLL-tot-star*:  
**assumes** *rough-state-of* (*DPLL-tot*  $S$ ) =  $S'$   
**shows** *dpll<sub>W</sub>\*\** (*toS'* (*rough-state-of*  $S$ )) (*toS'*  $S'$ )  
 $\langle \text{proof} \rangle$

**lemma** *rough-state-of-rough-state-of-Nil*[simp]:  
 $\text{rough-state-of } (\text{state-of } ([], N)) = ([], N)$   
 <proof>

Theorem of correctness

**lemma** *DPLL-tot-correct*:  
**assumes**  $\text{rough-state-of } (DPLL\text{-}tot (\text{state-of } ([], N))) = (M, N')$   
**and**  $(M', N'') = \text{toS}' (M, N')$   
**shows**  $M' \models_{asm} N'' \longleftrightarrow \text{satisfiable } (\text{set-mset } N'')$   
 <proof>

## Code export

**A conversion to DPLL-W-Implementation.** *dpll<sub>W</sub>-state*    **definition** *Con* :: (int, unit) ann-lits  $\times$  int literal list list

$\Rightarrow$  *dpll<sub>W</sub>-state* **where**

*Con xs* = *state-of* (if *dpll<sub>W</sub>-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]:  
 $\text{Con } (DPLL\text{-}step (\text{rough-state-of } s)) = \text{state-of } (DPLL\text{-}step (\text{rough-state-of } s))$   
 <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 ( $\forall A \in \text{set } N. (\exists a \in \text{set } A. a \in \text{lits-of-l } (M)), M$ ))

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 cdclW-mark` = `'a clause`

**type-synonym** `'v cdclW-ann-lit` = `('v, 'v cdclW-mark) ann-lit`

**type-synonym** `'v cdclW-ann-lits` = `('v, 'v cdclW-mark) ann-lits`

**type-synonym** `'v cdclW-state` =  
`'v cdclW-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 ≡ (λL (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 ≡ λk (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 ≡ λS (M, N, U, k, -). (M, N, U, k, S)`

**abbreviation** `S0-cdclW N ≡ ([], N, {#}, 0, None):: 'v cdclW-state`

**abbreviation** `raw-add-learned-clss` **where**

`raw-add-learned-clss ≡ λC (M, N, U, S). (M, N, {#C#} + U, S)`

**abbreviation** `raw-remove-cls` **where**

`raw-remove-cls ≡ λC (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*:

$S = (\text{raw-trail } S, \text{raw-init-clss } S, \text{raw-learned-clss } S, \text{raw-backtrack-lvl } S, \text{raw-conflicting } S)$   
 $\langle \text{proof} \rangle$

**interpretation** *state<sub>W</sub>*

*raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting*  
 $\lambda L (M, S). (L \# M, S)$   
 $\lambda (M, S). (tl \ M, S)$   
 $\lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$   
 $\lambda C (M, N, U, S). (M, \text{removeAll-mset } C \ N, \text{removeAll-mset } C \ U, S)$   
 $\lambda (k::nat) (M, N, U, -, D). (M, N, U, k, D)$   
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$   
 $\lambda N. ([], N, \{\#\}, 0, \text{None})$   
 $\langle \text{proof} \rangle$

**interpretation** *conflict-driven-clause-learning<sub>W</sub> raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting*

$\lambda L (M, S). (L \# M, S)$   
 $\lambda (M, S). (tl \ M, S)$   
 $\lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$   
 $\lambda C (M, N, U, S). (M, \text{removeAll-mset } C \ N, \text{removeAll-mset } C \ U, S)$   
 $\lambda (k::nat) (M, N, U, -, D). (M, N, U, k, D)$   
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$   
 $\lambda N. ([], N, \{\#\}, 0, \text{None})$   
 $\langle \text{proof} \rangle$

**declare** *clauses-def[simp]*

**lemma** *cdcl<sub>W</sub>-state-eq-equality[iff]*:  $\text{state-eq } S \ T \longleftrightarrow S = T$

$\langle \text{proof} \rangle$

**declare** *state-simp[simp del]*

**lemma** *reduce-trail-to-empty-trail[simp]*:

$\text{reduce-trail-to } F \ ([], aa, ab, ac, b) = ([], aa, ab, ac, b)$   
 $\langle \text{proof} \rangle$

**lemma** *raw-trail-reduce-trail-to-length-le*:

**assumes**  $\text{length } F > \text{length } (\text{raw-trail } S)$   
**shows**  $\text{raw-trail } (\text{reduce-trail-to } F \ S) = []$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-trail-to*:

$\text{reduce-trail-to } F \ S =$   
 $((\text{if } \text{length } (\text{raw-trail } S) \geq \text{length } F$   
 $\text{then drop } (\text{length } (\text{raw-trail } S) - \text{length } F) (\text{raw-trail } S)$   
 $\text{else } []), \text{raw-init-clss } S, \text{raw-learned-clss } S, \text{raw-backtrack-lvl } S, \text{raw-conflicting } S)$   
 $(\text{is } ?S = -)$   
 $\langle \text{proof} \rangle$

## 7.1.5 CDCL Implementation

### Definition of the rules

**Types** **lemma** *true-raw-init-clss-remdups[simp]*:

$I \models s \ (mset \circ \text{remdups}) \ ' \ N \longleftrightarrow I \models s \ mset \ ' \ N$



$\langle \text{proof} \rangle$

**lemma** *satisfiable-mset-remdups*[simp]:  
*satisfiable* ((*mset*  $\circ$  *remdups*) ‘ *N*)  $\longleftrightarrow$  *satisfiable* (*mset* ‘ *N*)  
 $\langle \text{proof} \rangle$

**type-synonym** ‘*v* *cdcl<sub>W</sub>-state-inv-st* = (‘*v*, ‘*v* *literal list*) *ann-lit list*  $\times$   
‘*v* *literal list list*  $\times$  ‘*v* *literal list list*  $\times$  *nat*  $\times$  ‘*v* *literal list option*

We need some functions to convert between our abstract state ‘*v* *cdcl<sub>W</sub>-state* and the concrete state ‘*v* *cdcl<sub>W</sub>-state-inv-st*.

**fun** *convert* :: (‘*a*, ‘*c* *list*) *ann-lit*  $\Rightarrow$  (‘*a*, ‘*c* *mset*) *ann-lit* **where**  
*convert* (*Propagated* *L* *C*) = *Propagated* *L* (*mset* *C*) |  
*convert* (*Decided* *K*) = *Decided* *K*

**abbreviation** *convertC* :: ‘*a* *list option*  $\Rightarrow$  ‘*a* *mset option* **where**  
*convertC*  $\equiv$  *map-option* *mset*

**lemma** *convert-Propagated*[elim!]:  
*convert* *z* = *Propagated* *L* *C*  $\Longrightarrow$  ( $\exists$  *C'*. *z* = *Propagated* *L* *C'*  $\wedge$  *C* = *mset* *C'*)  
 $\langle \text{proof} \rangle$

**lemma** *is-decided-convert*[simp]: *is-decided* (*convert* *x*) = *is-decided* *x*  
 $\langle \text{proof} \rangle$

**lemma** *get-level-map-convert*[simp]:  
*get-level* (*map* *convert* *M*) *x* = *get-level* *M* *x*  
 $\langle \text{proof} \rangle$

**lemma** *get-maximum-level-map-convert*[simp]:  
*get-maximum-level* (*map* *convert* *M*) *D* = *get-maximum-level* *M* *D*  
 $\langle \text{proof} \rangle$

Conversion function

**fun** *toS* :: ‘*v* *cdcl<sub>W</sub>-state-inv-st*  $\Rightarrow$  ‘*v* *cdcl<sub>W</sub>-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<sub>W</sub>-state-inv* = {*S*::‘*v* *cdcl<sub>W</sub>-state-inv-st*. *cdcl<sub>W</sub>-all-struct-inv* (*toS* *S*)}  
**morphisms** *rough-state-of* *state-of*  
 $\langle \text{proof} \rangle$

**instantiation** *cdcl<sub>W</sub>-state-inv* :: (*type*) *equal*  
**begin**

**definition** *equal-cdcl<sub>W</sub>-state-inv* :: ‘*v* *cdcl<sub>W</sub>-state-inv*  $\Rightarrow$  ‘*v* *cdcl<sub>W</sub>-state-inv*  $\Rightarrow$  *bool* **where**  
*equal-cdcl<sub>W</sub>-state-inv* *S* *S'* = (*rough-state-of* *S* = *rough-state-of* *S'*)

**instance**  
 $\langle \text{proof} \rangle$   
**end**

**lemma** *lits-of-map-convert*[simp]: *lits-of-l* (*map* *convert* *M*) = *lits-of-l* *M*  
 $\langle \text{proof} \rangle$

**lemma** *atm-lit-of-convert*[simp]:

*lit-of* (*convert* *x*) = *lit-of* *x*  
 ⟨*proof*⟩

**lemma** *undefined-lit-map-convert*[*iff*]:  
*undefined-lit* (*map convert* *M*) *L*  $\longleftrightarrow$  *undefined-lit* *M* *L*  
 ⟨*proof*⟩

**lemma** *true-annot-map-convert*[*simp*]: *map convert* *M*  $\models_a$  *N*  $\longleftrightarrow$  *M*  $\models_a$  *N*  
 ⟨*proof*⟩

**lemma** *true-annots-map-convert*[*simp*]: *map convert* *M*  $\models_{as}$  *N*  $\longleftrightarrow$  *M*  $\models_{as}$  *N*  
 ⟨*proof*⟩

**lemmas** *propagateE*

**lemma** *find-first-unit-clause-some-is-propagate*:

**assumes** *H*: *find-first-unit-clause* (*N* @ *U*) *M* = *Some* (*L*, *C*)

**shows** *propagate* (*toS* (*M*, *N*, *U*, *k*, *None*)) (*toS* (*Propagated* *L* *C* # *M*, *N*, *U*, *k*, *None*))

⟨*proof*⟩

## The Transitions

**Propagate** **definition** *do-propagate-step* **where**

*do-propagate-step* *S* =

(*case* *S* of

(*M*, *N*, *U*, *k*, *None*)  $\Rightarrow$

(*case* *find-first-unit-clause* (*N* @ *U*) *M* of

*Some* (*L*, *C*)  $\Rightarrow$  (*Propagated* *L* *C* # *M*, *N*, *U*, *k*, *None*)

| *None*  $\Rightarrow$  (*M*, *N*, *U*, *k*, *None*))

| *S*  $\Rightarrow$  *S*)

**lemma** *do-propagate-step*:

*do-propagate-step* *S*  $\neq$  *S*  $\implies$  *propagate* (*toS* *S*) (*toS* (*do-propagate-step* *S*))

⟨*proof*⟩

**lemma** *do-propagate-step-option*[*simp*]:

*raw-conflicting* *S*  $\neq$  *None*  $\implies$  *do-propagate-step* *S* = *S*

⟨*proof*⟩

**lemma** *do-propagate-step-no-step*:

**assumes** *dist*:  $\forall c \in \text{set } (\text{raw-init-clss } S @ \text{raw-learned-clss } S). \text{ distinct } c$  **and**

*prop-step*: *do-propagate-step* *S* = *S*

**shows** *no-step* *propagate* (*toS* *S*)

⟨*proof*⟩

**Conflict** **fun** *find-conflict* **where**

*find-conflict* *M* [] = *None* |

*find-conflict* *M* (*N* # *Ns*) = (*if* ( $\forall c \in \text{set } N. \neg c \in \text{lits-of-l } M$ ) *then* *Some* *N* *else* *find-conflict* *M* *Ns*)

**lemma** *find-conflict-Some*:

*find-conflict* *M* *Ns* = *Some* *N*  $\implies$  *N*  $\in$  *set* *Ns*  $\wedge$  *M*  $\models_{as}$  *CNot* (*mset* *N*)

⟨*proof*⟩

**lemma** *find-conflict-None*:

*find-conflict* *M* *Ns* = *None*  $\longleftrightarrow$  ( $\forall N \in \text{set } Ns. \neg M \models_{as} \text{CNot } (\text{mset } N)$ )

⟨*proof*⟩

**lemma** *find-conflict-None-no-conflict*:

*find-conflict*  $M (N @ U) = \text{None} \longleftrightarrow \text{no-step conflict } (\text{toS } (M, N, U, k, \text{None}))$   
 $\langle \text{proof} \rangle$

**definition** *do-conflict-step* **where**

*do-conflict-step*  $S =$

(*case*  $S$  of  
 ( $M, N, U, k, \text{None}$ )  $\Rightarrow$   
 (*case* *find-conflict*  $M (N @ U)$  of  
   *Some*  $a \Rightarrow (M, N, U, k, \text{Some } a)$   
   | *None*  $\Rightarrow (M, N, U, k, \text{None})$ )  
 |  $S \Rightarrow S$ )

**lemma** *do-conflict-step*:

*do-conflict-step*  $S \neq S \implies \text{conflict } (\text{toS } S) (\text{toS } (\text{do-conflict-step } S))$   
 $\langle \text{proof} \rangle$

**lemma** *do-conflict-step-no-step*:

*do-conflict-step*  $S = S \implies \text{no-step conflict } (\text{toS } S)$   
 $\langle \text{proof} \rangle$

**lemma** *do-conflict-step-option[simp]*:

*raw-conflicting*  $S \neq \text{None} \implies \text{do-conflict-step } S = S$   
 $\langle \text{proof} \rangle$

**lemma** *do-conflict-step-raw-conflicting[dest]*:

*do-conflict-step*  $S \neq S \implies \text{raw-conflicting } (\text{do-conflict-step } S) \neq \text{None}$   
 $\langle \text{proof} \rangle$

**definition** *do-cp-step* **where**

*do-cp-step*  $S =$

(*do-propagate-step*  $o$  *do-conflict-step*)  $S$

**lemma** *cp-step-is-cdcl<sub>W</sub>-cp*:

**assumes**  $H$ : *do-cp-step*  $S \neq S$   
**shows** *cdcl<sub>W</sub>-cp*  $(\text{toS } S) (\text{toS } (\text{do-cp-step } S))$   
 $\langle \text{proof} \rangle$

**lemma** *do-cp-step-eq-no-prop-no-conflict*:

*do-cp-step*  $S = S \implies \text{do-conflict-step } S = S \wedge \text{do-propagate-step } S = S$   
 $\langle \text{proof} \rangle$

**lemma** *no-cdcl<sub>W</sub>-cp-iff-no-propagate-no-conflict*:

*no-step cdcl<sub>W</sub>-cp*  $S \longleftrightarrow \text{no-step propagate } S \wedge \text{no-step conflict } S$   
 $\langle \text{proof} \rangle$

**lemma** *do-cp-step-eq-no-step*:

**assumes**  $H$ : *do-cp-step*  $S = S$  **and**  $\forall c \in \text{set } (\text{raw-init-clss } S @ \text{raw-learned-clss } S)$ . *distinct*  $c$   
**shows** *no-step cdcl<sub>W</sub>-cp*  $(\text{toS } S)$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-cdcl<sub>W</sub>-st*: *cdcl<sub>W</sub>-cp*  $S S' \implies \text{cdcl}_W^{**} S S'$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-all-struct-inv-rough-state[simp]*: *cdcl<sub>W</sub>-all-struct-inv*  $(\text{toS } (\text{rough-state-of } S))$

$\langle \text{proof} \rangle$

**lemma** [simp]:  $cdcl_W\text{-all-struct-inv } (toS\ S) \implies \text{rough-state-of } (state\text{-of } S) = S$   
 ⟨proof⟩

**lemma** *rough-state-of-state-of-do-cp-step*[simp]:  
 $\text{rough-state-of } (state\text{-of } (do\text{-cp-step } (\text{rough-state-of } S))) = do\text{-cp-step } (\text{rough-state-of } S)$   
 ⟨proof⟩

**Skip fun** *do-skip-step* ::  $'v\ cdcl_W\text{-state-inv-st} \Rightarrow 'v\ cdcl_W\text{-state-inv-st}$  **where**  
*do-skip-step* (Propagated  $L\ C\ \# Ls, N, U, k, Some\ D$ ) =  
 (if  $-L \notin \text{set } D \wedge D \neq []$   
 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*:  
 $do\text{-skip-step } S \neq S \implies \text{skip } (toS\ S) (toS\ (do\text{-skip-step } S))$   
 ⟨proof⟩

**lemma** *do-skip-step-no*:  
 $do\text{-skip-step } S = S \implies \text{no-step skip } (toS\ S)$   
 ⟨proof⟩

**lemma** *do-skip-step-raw-trail-is-None*[iff]:  
 $do\text{-skip-step } S = (a, b, c, d, None) \longleftrightarrow S = (a, b, c, d, None)$   
 ⟨proof⟩

**Resolve fun** *maximum-level-code*::  $'a\ \text{literal list} \Rightarrow ('a, 'a\ \text{literal list})\ \text{ann-lit list} \Rightarrow \text{nat}$   
**where**  
*maximum-level-code* [] = 0 |  
*maximum-level-code* ( $L\ \# Ls$ )  $M = \max\ (\text{get-level } M\ L)\ (\text{maximum-level-code } Ls\ M)$

**lemma** *maximum-level-code-eq-get-maximum-level*[code, simp]:  
 $\text{maximum-level-code } D\ M = \text{get-maximum-level } M\ (\text{mset } D)$   
 ⟨proof⟩

**fun** *do-resolve-step* ::  $'v\ cdcl_W\text{-state-inv-st} \Rightarrow 'v\ cdcl_W\text{-state-inv-st}$  **where**  
*do-resolve-step* (Propagated  $L\ C\ \# Ls, N, U, k, Some\ D$ ) =  
 (if  $-L \in \text{set } D \wedge \text{maximum-level-code } (\text{remove1 } (-L)\ D) (\text{Propagated } L\ C\ \# Ls) = k$   
 then  $(Ls, N, U, k, Some\ (\text{remdups } (\text{remove1 } L\ C\ @\ \text{remove1 } (-L)\ D)))$   
 else (Propagated  $L\ C\ \# Ls, N, U, k, Some\ D$ )) |  
*do-resolve-step*  $S = S$

**lemma** *do-resolve-step*:  
 $cdcl_W\text{-all-struct-inv } (toS\ S) \implies do\text{-resolve-step } S \neq S$   
 $\implies \text{resolve } (toS\ S) (toS\ (do\text{-resolve-step } S))$   
 ⟨proof⟩

**lemma** *do-resolve-step-no*:  
 $do\text{-resolve-step } S = S \implies \text{no-step resolve } (toS\ S)$   
 ⟨proof⟩

**lemma** *rough-state-of-state-of-resolve*[simp]:  
 $cdcl_W\text{-all-struct-inv } (toS\ S) \implies \text{rough-state-of } (state\text{-of } (do\text{-resolve-step } S)) = do\text{-resolve-step } S$   
 ⟨proof⟩

**lemma** *do-resolve-step-raw-trail-is-None*[iff]:  
 $do\_resolve\_step\ S = (a, b, c, d, None) \longleftrightarrow S = (a, b, c, d, None)$   
 $\langle proof \rangle$

**Backjumping lemma** *get-all-ann-decomposition-map-convert*:  
 $(get\_all\_ann\_decomposition\ (map\ convert\ M)) =$   
 $map\ (\lambda(a, b). (map\ convert\ a, map\ convert\ b))\ (get\_all\_ann\_decomposition\ M)$   
 $\langle proof \rangle$

**lemma** *do-backtrack-step*:  
**assumes**  
 $db: do\_backtrack\_step\ S \neq S$  **and**  
 $inv: cdcl_W\text{-all-struct-inv}\ (toS\ S)$   
**shows**  $backtrack\ (toS\ S)\ (toS\ (do\_backtrack\_step\ S))$   
 $\langle proof \rangle$

**lemma** *map-eq-list-length*:  
 $map\ f\ L = L' \implies length\ L = length\ L'$   
 $\langle proof \rangle$

**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**  
 $c = map\ convert\ c'$   
 $\langle proof \rangle$

**lemma** *Decided-convert-iff*:  
 $Decided\ K = convert\ za \longleftrightarrow za = Decided\ K$   
 $\langle proof \rangle$

**lemma** *do-backtrack-step-no*:  
**assumes**  
 $db: do\_backtrack\_step\ S = S$  **and**  
 $inv: cdcl_W\text{-all-struct-inv}\ (toS\ S)$   
**shows**  $no\_step\ backtrack\ (toS\ S)$   
 $\langle proof \rangle$

**lemma** *rough-state-of-state-of-backtrack*[simp]:  
**assumes**  $inv: cdcl_W\text{-all-struct-inv}\ (toS\ S)$   
**shows**  $rough\_state\_of\ (state\_of\ (do\_backtrack\_step\ S)) = do\_backtrack\_step\ S$   
 $\langle proof \rangle$

**Decide fun** *do-decide-step* **where**  
 $do\_decide\_step\ (M, N, U, k, None) =$   
 $(case\ find\_first\_unused\_var\ N\ (lits\_of\_l\ M)\ of$   
 $None \Rightarrow (M, N, U, k, None)$   
 $| Some\ L \Rightarrow (Decided\ L\ \# \ M, N, U, k+1, None))\ |$   
 $do\_decide\_step\ S = S$

**lemma** *do-decide-step*:  
 $do\_decide\_step\ S \neq S \implies decide\ (toS\ S)\ (toS\ (do\_decide\_step\ S))$   
 $\langle proof \rangle$

**lemma** *do-decide-step-no*:

*do-decide-step S = S  $\implies$  no-step decide (toS S)*

*<proof>*

**lemma** *rough-state-of-state-of-do-decide-step[simp]*:

*cdcl<sub>W</sub>-all-struct-inv (toS S)  $\implies$  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<sub>W</sub>-all-struct-inv (toS S)  $\implies$  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<sub>W</sub>-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<sub>W</sub>-state-inv-from-init-state =*

*{S:: 'v cdcl<sub>W</sub>-state-inv-st. cdcl<sub>W</sub>-all-struct-inv (toS S)  
   $\wedge$  cdcl<sub>W</sub>-stgy\*\* (S0-cdcl<sub>W</sub> (raw-init-clss (toS S))) (toS S)}*

**morphisms** *rough-state-from-init-state-of state-from-init-state-of*

*<proof>*

**instantiation** *cdcl<sub>W</sub>-state-inv-from-init-state :: (type) equal*

**begin**

**definition** *equal-cdcl<sub>W</sub>-state-inv-from-init-state :: 'v cdcl<sub>W</sub>-state-inv-from-init-state  $\Rightarrow$*

*'v cdcl<sub>W</sub>-state-inv-from-init-state  $\Rightarrow$  bool* **where**

*equal-cdcl<sub>W</sub>-state-inv-from-init-state S S'  $\longleftrightarrow$*

*(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<sub>W</sub>-all-struct-inv (toS (fst S, snd S))  
   $\wedge$  cdcl<sub>W</sub>-stgy\*\* (S0-cdcl<sub>W</sub> (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<sub>W</sub>-state-inv-from-init-state  $\Rightarrow$  'v cdcl<sub>W</sub>-state-inv* **where**

$id\text{-}of\text{-}I\text{-}to\ S = state\text{-}of\ (rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S)$

**lemma** *[code abstract]:*

$rough\text{-}state\text{-}of\ (id\text{-}of\text{-}I\text{-}to\ S) = rough\text{-}state\text{-}from\text{-}init\text{-}state\text{-}of\ S$   
 $\langle proof \rangle$

**Conflict and Propagate function**  $do\text{-}full1\text{-}cp\text{-}step :: 'v\ cdcl_W\text{-}state\text{-}inv \Rightarrow 'v\ cdcl_W\text{-}state\text{-}inv$   
**where**

$do\text{-}full1\text{-}cp\text{-}step\ S =$   
 $(let\ S' = do\text{-}cp\text{-}step'\ S\ in$   
 $\quad if\ S = S'\ then\ S\ else\ do\text{-}full1\text{-}cp\text{-}step\ S')$   
 $\langle proof \rangle$

**termination**

$\langle proof \rangle$

**lemma**  $do\text{-}full1\text{-}cp\text{-}step\text{-}fix\text{-}point\text{-}of\text{-}do\text{-}full1\text{-}cp\text{-}step$ :

$do\text{-}cp\text{-}step(rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S)) = (rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S))$   
 $\langle proof \rangle$

**lemma**  $in\text{-}clauses\text{-}rough\text{-}state\text{-}of\text{-}is\text{-}distinct$ :

$c \in set\ (raw\text{-}init\text{-}clss\ (rough\text{-}state\text{-}of\ S) @ raw\text{-}learned\text{-}clss\ (rough\text{-}state\text{-}of\ S)) \implies distinct\ c$   
 $\langle proof \rangle$

**lemma**  $do\text{-}full1\text{-}cp\text{-}step\text{-}full$ :

$full\ cdcl_W\text{-}cp\ (toS\ (rough\text{-}state\text{-}of\ S))$   
 $(toS\ (rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S)))$   
 $\langle proof \rangle$

**lemma** *[code abstract]:*

$rough\text{-}state\text{-}of\ (do\text{-}cp\text{-}step'\ S) = do\text{-}cp\text{-}step\ (rough\text{-}state\text{-}of\ S)$   
 $\langle proof \rangle$

**The other rules** **fun**  $do\text{-}other\text{-}step$  **where**

$do\text{-}other\text{-}step\ S =$   
 $(let\ T = do\text{-}skip\text{-}step\ S\ in$   
 $\quad if\ T \neq S$   
 $\quad then\ T$   
 $\quad else$   
 $\quad (let\ U = do\text{-}resolve\text{-}step\ T\ in$   
 $\quad \quad if\ U \neq T$   
 $\quad \quad then\ U\ else$   
 $\quad \quad (let\ V = do\text{-}backtrack\text{-}step\ U\ in$   
 $\quad \quad \quad if\ V \neq U\ then\ V\ else\ do\text{-}decide\text{-}step\ V)))$

**lemma**  $do\text{-}other\text{-}step$ :

**assumes**  $inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ S)$  **and**  
 $st: do\text{-}other\text{-}step\ S \neq S$   
**shows**  $cdcl_W\text{-}o\ (toS\ S)\ (toS\ (do\text{-}other\text{-}step\ S))$   
 $\langle proof \rangle$

**lemma**  $do\text{-}other\text{-}step\text{-}no$ :

**assumes**  $inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ (toS\ S)$  **and**  
 $st: do\text{-}other\text{-}step\ S = S$   
**shows**  $no\text{-}step\ cdcl_W\text{-}o\ (toS\ S)$   
 $\langle proof \rangle$

**lemma** *rough-state-of-state-of-do-other-step*[simp]:  
 $\text{rough-state-of } (\text{state-of } (\text{do-other-step } (\text{rough-state-of } S))) = \text{do-other-step } (\text{rough-state-of } S)$   
 ⟨proof⟩

**definition** *do-other-step'* **where**  
 $\text{do-other-step}' S =$   
 $\text{state-of } (\text{do-other-step } (\text{rough-state-of } S))$

**lemma** *rough-state-of-do-other-step'*[code abstract]:  
 $\text{rough-state-of } (\text{do-other-step}' S) = \text{do-other-step } (\text{rough-state-of } S)$   
 ⟨proof⟩

**definition** *do-cdcl<sub>W</sub>-stgy-step* **where**  
 $\text{do-cdcl}_W\text{-stgy-step } S =$   
 $(\text{let } T = \text{do-full1-cp-step } S \text{ in}$   
 $\text{if } T \neq S$   
 $\text{then } T$   
 $\text{else}$   
 $(\text{let } U = (\text{do-other-step}' T) \text{ in}$   
 $(\text{do-full1-cp-step } U)))$

**definition** *do-cdcl<sub>W</sub>-stgy-step'* **where**  
 $\text{do-cdcl}_W\text{-stgy-step}' S = \text{state-from-init-state-of } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{id-of-I-to } S)))$

**lemma** *toS-do-full1-cp-step-not-eq*:  $\text{do-full1-cp-step } S \neq S \implies$   
 $\text{toS } (\text{rough-state-of } S) \neq \text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } S))$   
 ⟨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<sub>W</sub>-stgy-step*:  
**assumes**  $\text{do-cdcl}_W\text{-stgy-step } S \neq S$   
**shows**  $\text{cdcl}_W\text{-stgy } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } S)))$   
 ⟨proof⟩

**lemma** *length-raw-trail-toS*[simp]:  
 $\text{length } (\text{raw-trail } (\text{toS } S)) = \text{length } (\text{raw-trail } S)$   
 ⟨proof⟩

**lemma** *raw-conflicting-noTrue-iff-toS*[simp]:  
 $\text{raw-conflicting } (\text{toS } S) \neq \text{None} \longleftrightarrow \text{raw-conflicting } S \neq \text{None}$   
 ⟨proof⟩

**lemma** *raw-trail-toS-neq-imp-raw-trail-neq*:  
 $\text{raw-trail } (\text{toS } S) \neq \text{raw-trail } (\text{toS } S') \implies \text{raw-trail } S \neq \text{raw-trail } S'$   
 ⟨proof⟩

**lemma** *do-skip-step-raw-trail-changed-or-conflict*:  
**assumes**  $d: \text{do-other-step } S \neq S$   
**and**  $\text{inv}: \text{cdcl}_W\text{-all-struct-inv } (\text{toS } S)$   
**shows**  $\text{raw-trail } S \neq \text{raw-trail } (\text{do-other-step } S)$   
 ⟨proof⟩

**lemma** *do-full1-cp-step-induct*:



$(\bigwedge S. (S \neq \text{do-cp-step}' S \implies P (\text{do-cp-step}' S)) \implies P S) \implies P a0$   
 $\langle \text{proof} \rangle$

**lemma** *do-cp-step-neq-raw-trail-increase:*

$\exists c. \text{raw-trail } (\text{do-cp-step } S) = c @ \text{raw-trail } S \wedge (\forall m \in \text{set } c. \neg \text{is-decided } m)$   
 $\langle \text{proof} \rangle$

**lemma** *do-full1-cp-step-neq-raw-trail-increase:*

$\exists c. \text{raw-trail } (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{raw-trail } (\text{rough-state-of } S)$   
 $\wedge (\forall m \in \text{set } c. \neg \text{is-decided } m)$   
 $\langle \text{proof} \rangle$

**lemma** *do-cp-step-raw-conflicting:*

$\text{raw-conflicting } (\text{rough-state-of } S) \neq \text{None} \implies \text{do-cp-step}' S = S$   
 $\langle \text{proof} \rangle$

**lemma** *do-full1-cp-step-raw-conflicting:*

$\text{raw-conflicting } (\text{rough-state-of } S) \neq \text{None} \implies \text{do-full1-cp-step } S = S$   
 $\langle \text{proof} \rangle$

**lemma** *do-decide-step-not-raw-conflicting-one-more-decide:*

**assumes**  
 $\text{raw-conflicting } S = \text{None}$  **and**  
 $\text{do-decide-step } S \neq S$   
**shows**  $\text{Suc } (\text{length } (\text{filter is-decided } (\text{raw-trail } S)))$   
 $= \text{length } (\text{filter is-decided } (\text{raw-trail } (\text{do-decide-step } S)))$   
 $\langle \text{proof} \rangle$

**lemma** *do-decide-step-not-raw-conflicting-one-more-decide-bt:*

**assumes**  $\text{raw-conflicting } S \neq \text{None}$  **and**  
 $\text{do-decide-step } S \neq S$   
**shows**  $\text{length } (\text{filter is-decided } (\text{raw-trail } S)) < \text{length } (\text{filter is-decided } (\text{raw-trail } (\text{do-decide-step } S)))$   
 $\langle \text{proof} \rangle$

**lemma** *count-decided-raw-trail-toS:*

$\text{count-decided } (\text{raw-trail } (\text{toS } S)) = \text{count-decided } (\text{raw-trail } S)$   
 $\langle \text{proof} \rangle$

**lemma** *do-other-step-not-raw-conflicting-one-more-decide-bt:*

**assumes**  
 $\text{raw-conflicting } (\text{rough-state-of } S) \neq \text{None}$  **and**  
 $\text{raw-conflicting } (\text{rough-state-of } (\text{do-other-step}' S)) = \text{None}$  **and**  
 $\text{do-other-step}' S \neq S$   
**shows**  $\text{count-decided } (\text{raw-trail } (\text{rough-state-of } S))$   
 $> \text{count-decided } (\text{raw-trail } (\text{rough-state-of } (\text{do-other-step}' S)))$   
 $\langle \text{proof} \rangle$

**lemma** *do-other-step-not-raw-conflicting-one-more-decide:*

**assumes**  $\text{raw-conflicting } (\text{rough-state-of } S) = \text{None}$  **and**  
 $\text{do-other-step}' S \neq S$   
**shows**  $1 + \text{length } (\text{filter is-decided } (\text{raw-trail } (\text{rough-state-of } S)))$   
 $= \text{length } (\text{filter is-decided } (\text{raw-trail } (\text{rough-state-of } (\text{do-other-step}' S))))$   
 $\langle \text{proof} \rangle$

**lemma** *rough-state-of-state-of-do-skip-step-rough-state-of[simp]:*

$\text{rough-state-of } (\text{state-of } (\text{do-skip-step } (\text{rough-state-of } S))) = \text{do-skip-step } (\text{rough-state-of } S)$

$\langle \text{proof} \rangle$

**lemma** *raw-conflicting-do-resolve-step-iff*[iff]:

$\text{raw-conflicting } (\text{do-resolve-step } S) = \text{None} \longleftrightarrow \text{raw-conflicting } S = \text{None}$

$\langle \text{proof} \rangle$

**lemma** *raw-conflicting-do-skip-step-iff*[iff]:

$\text{raw-conflicting } (\text{do-skip-step } S) = \text{None} \longleftrightarrow \text{raw-conflicting } S = \text{None}$

$\langle \text{proof} \rangle$

**lemma** *raw-conflicting-do-decide-step-iff*[iff]:

$\text{raw-conflicting } (\text{do-decide-step } S) = \text{None} \longleftrightarrow \text{raw-conflicting } S = \text{None}$

$\langle \text{proof} \rangle$

**lemma** *raw-conflicting-do-backtrack-step-imp*[simp]:

$\text{do-backtrack-step } S \neq S \implies \text{raw-conflicting } (\text{do-backtrack-step } S) = \text{None}$

$\langle \text{proof} \rangle$

**lemma** *do-skip-step-eq-iff-raw-trail-eq*:

$\text{do-skip-step } S = S \longleftrightarrow \text{raw-trail } (\text{do-skip-step } S) = \text{raw-trail } S$

$\langle \text{proof} \rangle$

**lemma** *do-decide-step-eq-iff-raw-trail-eq*:

$\text{do-decide-step } S = S \longleftrightarrow \text{raw-trail } (\text{do-decide-step } S) = \text{raw-trail } S$

$\langle \text{proof} \rangle$

**lemma** *do-backtrack-step-eq-iff-raw-trail-eq*:

**assumes** *no-dup* (*raw-trail* *S*)

**shows**  $\text{do-backtrack-step } S = S \longleftrightarrow \text{raw-trail } (\text{do-backtrack-step } S) = \text{raw-trail } S$

$\langle \text{proof} \rangle$

**lemma** *do-resolve-step-eq-iff-raw-trail-eq*:

$\text{do-resolve-step } S = S \longleftrightarrow \text{raw-trail } (\text{do-resolve-step } S) = \text{raw-trail } S$

$\langle \text{proof} \rangle$

**lemma** *do-other-step-eq-iff-raw-trail-eq*:

**assumes** *no-dup* (*raw-trail* *S*)

**shows**  $\text{raw-trail } (\text{do-other-step } S) = \text{raw-trail } S \longleftrightarrow \text{do-other-step } S = S$

$\langle \text{proof} \rangle$

**lemma** *do-full1-cp-step-do-other-step'-normal-form*[dest!]:

**assumes** *H*:  $\text{do-full1-cp-step } (\text{do-other-step}' S) = S$

**shows**  $\text{do-other-step}' S = S \wedge \text{do-full1-cp-step } S = S$

$\langle \text{proof} \rangle$

**lemma** *do-cdcl<sub>W</sub>-stgy-step-no*:

**assumes** *S*:  $\text{do-cdcl}_W\text{-stgy-step } S = S$

**shows**  $\text{no-step } \text{cdcl}_W\text{-stgy } (\text{toS } (\text{rough-state-of } S))$

$\langle \text{proof} \rangle$

**lemma** *toS-rough-state-of-state-of-rough-state-from-init-state-of*[simp]:

$\text{toS } (\text{rough-state-of } (\text{state-of } (\text{rough-state-from-init-state-of } S)))$

$= \text{toS } (\text{rough-state-from-init-state-of } S)$

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-cp-is-rtrancpl-cdcl<sub>W</sub>*: *cdcl<sub>W</sub>-cp S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*  
 ⟨proof⟩

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-is-rtrancpl-cdcl<sub>W</sub>*: *cdcl<sub>W</sub>-cp<sup>\*\*</sup> S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*  
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-stgy-is-rtrancpl-cdcl<sub>W</sub>*:  
*cdcl<sub>W</sub>-stgy S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*  
 ⟨proof⟩

**lemma** *cdcl<sub>W</sub>-stgy-init-raw-init-clss*:  
*cdcl<sub>W</sub>-stgy S T  $\implies$  cdcl<sub>W</sub>-M-level-inv S  $\implies$  raw-init-clss S = raw-init-clss T*  
 ⟨proof⟩

**lemma** *clauses-toS-rough-state-of-do-cdcl<sub>W</sub>-stgy-step[simp]*:  
*raw-init-clss (toS (rough-state-of (do-cdcl<sub>W</sub>-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<sub>W</sub>-stgy-step'[code abstract]*:  
*rough-state-from-init-state-of (do-cdcl<sub>W</sub>-stgy-step' S) =*  
*rough-state-of (do-cdcl<sub>W</sub>-stgy-step (id-of-I-to S))*  
 ⟨proof⟩

**All rules together function do-all-cdcl<sub>W</sub>-stgy where**

*do-all-cdcl<sub>W</sub>-stgy S =*  
*(let T = do-cdcl<sub>W</sub>-stgy-step' S in*  
*if T = S then S else do-all-cdcl<sub>W</sub>-stgy T)*  
 ⟨proof⟩

**termination**  
 ⟨proof⟩

**thm** *do-all-cdcl<sub>W</sub>-stgy.induct*

**lemma** *do-all-cdcl<sub>W</sub>-stgy.induct*:  
*( $\bigwedge S. (do-cdcl<sub>W</sub>-stgy-step' S \neq S \implies P (do-cdcl<sub>W</sub>-stgy-step' S)) \implies P S) \implies P a0$*   
 ⟨proof⟩

**lemma** *no-step-cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-all*:  
**fixes** *S :: 'a cdcl<sub>W</sub>-state-inv-from-init-state*  
**shows** *no-step cdcl<sub>W</sub>-stgy (toS (rough-state-from-init-state-of (do-all-cdcl<sub>W</sub>-stgy S)))*  
 ⟨proof⟩

**lemma** *do-all-cdcl<sub>W</sub>-stgy-is-rtrancpl-cdcl<sub>W</sub>-stgy*:  
*cdcl<sub>W</sub>-stgy<sup>\*\*</sup> (toS (rough-state-from-init-state-of S))*  
*(toS (rough-state-from-init-state-of (do-all-cdcl<sub>W</sub>-stgy S)))*  
 ⟨proof⟩

Final theorem:

**lemma** *DPLL-tot-correct*:

**assumes**

*r: rough-state-from-init-state-of (do-all-cdcl<sub>W</sub>-stgy (state-from-init-state-of*  
*(([], map remdups N, [], 0, None)))) = S and*

*S: (M', N', U', k, E) = toS S*

**shows** *(E  $\neq$  Some {#}  $\wedge$  satisfiable (set (map mset N)))*  
 *$\vee$  (E = Some {#}  $\wedge$  unsatisfiable (set (map mset N)))*

*<proof>*

**The Code** The SML code is skipped in the documentation, but stays to ensure that some version of the exported code is working. The only difference between the generated code and the one used here is the export of the constructor `ConL`.

```
end
theory CDCL-Abstract-Clause-Representation
imports Main Partial-Clausal-Logic
begin

type-synonym 'v clause = 'v literal multiset
type-synonym 'v clauses = 'v clause multiset
```

### 7.1.6 Abstract Clause Representation

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

We assume the following:

- there is an equivalent to adding and removing a literal and to taking the union of clauses.

```
locale raw-cls =
  fixes
    mset-cls :: 'cls  $\Rightarrow$  'v clause
begin
end

locale raw-ccls-union =
  fixes
    mset-cls :: 'cls  $\Rightarrow$  'v clause and
    union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
    remove-clit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls
  assumes
    mset-ccls-union-cl[simp]: mset-cls (union-cls C D) = mset-cls C # $\cup$  mset-cls D and
    remove-clit[simp]: mset-cls (remove-clit L C) = remove1-mset L (mset-cls C)
begin
end
```

Instantiation of the previous locale, in an unnamed context to avoid polluting with simp rules

```
context
begin
  interpretation list-cls: raw-cls mset
    <proof>

  interpretation cls-cls: raw-cls id
    <proof>

  interpretation list-cls: raw-ccls-union mset
    union-mset-list remove1
    <proof>

  interpretation cls-cls: raw-ccls-union id op # $\cup$  remove1-mset
```

$\langle proof \rangle$   
**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-clss* *mset-clss*  
**for**  
*mset-clss* :: 'cls  $\Rightarrow$  'v clause +  
**fixes**  
*mset-clss*:: 'clss  $\Rightarrow$  'v clauses **and**  
*union-clss* :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss **and**  
*in-clss* :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool **and**  
*insert-clss* :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss **and**  
*remove-from-clss* :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss  
**assumes**  
*insert-clss*[simp]: *mset-clss* (*insert-clss* L C) = *mset-clss* C + {#*mset-clss* L#} **and**  
*union-clss*[simp]: *mset-clss* (*union-clss* C D) = *mset-clss* C + *mset-clss* D **and**  
*mset-clss-union-clss*[simp]: *mset-clss* (*insert-clss* C' D) = {#*mset-clss* C'#} + *mset-clss* D **and**  
*in-clss-mset-clss*[dest]: *in-clss* a C  $\implies$  *mset-clss* a  $\in$  # *mset-clss* C **and**  
*in-mset-clss-exists-preimage*: *b*  $\in$  # *mset-clss* C  $\implies$   $\exists$  b'. *in-clss* b' C  $\wedge$  *mset-clss* b' = b **and**  
*remove-from-clss-mset-clss*[simp]:  
*mset-clss* (*remove-from-clss* a C) = *mset-clss* C - {#*mset-clss* a#} **and**  
*in-clss-union-clss*[simp]:  
*in-clss* a (*union-clss* C D)  $\longleftrightarrow$  *in-clss* a C  $\vee$  *in-clss* a D  
**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))  
 $\langle proof \rangle$   
  
**interpretation** *clss-clss*: *raw-clss* *id*  
*id* op + op  $\in$  #  $\lambda$ L C. C + {#L#} *remove1-mset*  
 $\langle proof \rangle$   
  
**interpretation** *list-clss*: *raw-clss* *mset*  
 $\lambda$ L. *mset* (map *mset* L) op @  $\lambda$ L C. L  $\in$  set C op #  
*remove-first*  
 $\langle proof \rangle$   
**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 cdclW-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 cdclW-mset ⇒ ('v, 'v clause) ann-lit list where
trail ≡ λ(M, -). M

```

```

definition init-clss :: 'v cdclW-mset ⇒ 'v clauses where
init-clss ≡ λ(-, N, -). N

```

```

definition learned-clss :: 'v cdclW-mset ⇒ 'v clauses where
learned-clss ≡ λ(-, -, U, -). U

```

```

definition backtrack-lvl :: 'v cdclW-mset ⇒ nat where
backtrack-lvl ≡ λ(-, -, -, k, -). k

```

```

definition conflicting :: 'v cdclW-mset ⇒ 'v clause option where
conflicting ≡ λ(-, -, -, -, C). C

```

```

definition cons-trail :: ('v, 'v clause) ann-lit ⇒ 'v cdclW-mset ⇒ 'v cdclW-mset where
cons-trail ≡ λL (M, R). (L # M, R)

```

```

definition tl-trail where
tl-trail ≡ λ(M, R). (tl M, R)

```

```

definition add-learned-clss where
add-learned-clss ≡ λC (M, N, U, R). (M, N, {#C#} + U, R)

```

```

definition remove-clss where
remove-clss ≡ λC (M, N, U, R). (M, removeAll-mset C N, removeAll-mset C U, R)

```

```

definition update-backtrack-lvl where
update-backtrack-lvl ≡ λk (M, N, U, -, D). (M, N, U, k, D)

```

```

definition update-conflicting where
update-conflicting ≡ λD (M, N, U, k, -). (M, N, U, k, D)

```

**definition** *init-state* **where**

*init-state*  $\equiv \lambda N. ([], N, \{\#\}, 0, None)$

**lemmas** *cdcl<sub>W</sub>-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<sub>W</sub>-mset: state<sub>W</sub>-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*

$\langle \text{proof} \rangle$

**interpretation** *cdcl<sub>W</sub>-mset: state<sub>W</sub>* **where**

*trail* = *trail* **and**

*init-clss* = *init-clss* **and**

*learned-clss* = *learned-clss* **and**

*backtrack-lvl* = *backtrack-lvl* **and**

*conflicting* = *conflicting* **and**

*cons-trail* = *cons-trail* **and**

*tl-trail* = *tl-trail* **and**

*add-learned-cls* = *add-learned-cls* **and**

*remove-cls* = *remove-cls* **and**

*update-backtrack-lvl* = *update-backtrack-lvl* **and**

*update-conflicting* = *update-conflicting* **and**

*init-state* = *init-state*

$\langle \text{proof} \rangle$

**interpretation** *cdcl<sub>W</sub>-mset: conflict-driven-clause-learning<sub>W</sub>* **where**

*trail* = *trail* **and**

*init-clss* = *init-clss* **and**

*learned-clss* = *learned-clss* **and**

*backtrack-lvl* = *backtrack-lvl* **and**

*conflicting* = *conflicting* **and**

*cons-trail* = *cons-trail* **and**

*tl-trail* = *tl-trail* **and**

*add-learned-cls* = *add-learned-cls* **and**

*remove-cls* = *remove-cls* **and**

*update-backtrack-lvl* = *update-backtrack-lvl* **and**

*update-conflicting* = *update-conflicting* **and**

*init-state* = *init-state*

$\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-mset-state-eq-eq*: *cdcl<sub>W</sub>-mset.state-eq* = (*op* =)  
 ⟨*proof*⟩

**notation** *cdcl<sub>W</sub>-mset.state-eq* (**infix**  $\sim_m$  49)

## 7.2.2 Abstract Relation and Relation Theorems

This locale makes the lifting from the relation defined with multiset  $R$  and the version with an abstract state  $R\text{-abs}$ . We are lifting many different relations (each rule and the the strategy).

**locale** *relation-implied-relation-abs* =

**fixes**

$R :: 'v \text{ cdcl}_W\text{-mset} \Rightarrow 'v \text{ cdcl}_W\text{-mset} \Rightarrow \text{bool}$  **and**

$R\text{-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **and**

$\text{state} :: 'st \Rightarrow 'v \text{ cdcl}_W\text{-mset}$  **and**

$\text{inv} :: 'v \text{ cdcl}_W\text{-mset} \Rightarrow \text{bool}$

**assumes**

*relation-compatible-state*:

$\text{inv} (\text{state } S) \Longrightarrow R\text{-abs } S \ T \Longrightarrow R (\text{state } S) (\text{state } T)$  **and**

*relation-compatible-abs*:

$\bigwedge S \ S' \ T. \text{inv } S \Longrightarrow S \sim_m \text{state } S' \Longrightarrow R \ S \ T \Longrightarrow \exists U. R\text{-abs } S' \ U \wedge T \sim_m \text{state } U$  **and**

*relation-invariant*:

$\bigwedge S \ T. R \ S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$  **and**

*relation-abs-right-compatible*:

$\bigwedge S \ T \ U. \text{inv} (\text{state } S) \Longrightarrow R\text{-abs } S \ T \Longrightarrow \text{state } T \sim_m \text{state } U \Longrightarrow R\text{-abs } S \ U$

**begin**

**lemma** *relation-compatible-eq*:

**assumes**

$\text{inv}: \text{inv} (\text{state } S)$  **and**

$\text{abs}: R\text{-abs } S \ T$  **and**

$SS': \text{state } S \sim_m \text{state } S'$  **and**

$TT': \text{state } T \sim_m \text{state } T'$

**shows**  $R\text{-abs } S' \ T'$

⟨*proof*⟩

**lemma** *rtrancp-relation-invariant*:

$R^{++} \ S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$

⟨*proof*⟩

**lemma** *rtrancp-abs-rtrancp*:

$R\text{-abs}^{**} \ S \ T \Longrightarrow \text{inv} (\text{state } S) \Longrightarrow R^{**} (\text{state } S) (\text{state } T)$

⟨*proof*⟩

**lemma** *trancp-relation-trancp-relation-abs-compatible*:

**fixes**  $S :: 'st$

**assumes**

$R: R^{++} (\text{state } S) \ T$  **and**

$\text{inv}: \text{inv} (\text{state } S)$

**shows**  $\exists U. R\text{-abs}^{++} \ S \ U \wedge T \sim_m \text{state } U$

⟨*proof*⟩

**lemma** *rtrancp-relation-rtrancp-relation-abs-compatible*:

**fixes**  $S :: 'st$

**assumes**



$R: R^{**} (state\ S)\ T$  **and**  
 $inv: inv\ (state\ S)$   
**shows**  $\exists U. R-abs^{**}\ S\ U \wedge T \sim_m state\ U$   
 $\langle proof \rangle$

**lemma** *no-step-iff*:  
 $inv\ (state\ S) \implies no-step\ R\ (state\ S) \longleftrightarrow no-step\ R-abs\ S$   
 $\langle proof \rangle$

**lemma** *trancpl-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**  $R-abs^{++}\ S'\ U \wedge inv\ (state\ U)$   
 $\langle proof \rangle$

**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**  
*trancpl-relation-compatible-eq*:  $R-abs^{++}\ S'\ U$  **and**  
*trancpl-relation-abs-invariant*:  $inv\ (state\ U)$   
 $\langle proof \rangle$

**lemma** *trancpl-abs-trancpl*:  $R-abs^{++}\ S\ T \implies inv\ (state\ S) \implies R^{++}\ (state\ S)\ (state\ T)$   
 $\langle proof \rangle$

**lemma** *full1-iff*:

**assumes**  $inv: inv\ (state\ S)$   
**shows**  $full1\ R\ (state\ S)\ (state\ T) \longleftrightarrow full1\ R-abs\ S\ T$  (**is**  $?R \longleftrightarrow ?R-abs$ )  
 $\langle proof \rangle$

**lemma** *full1-iff-compatible*:

**assumes**  $inv: inv\ (state\ S)$  **and**  $SS': S' \sim_m state\ S$  **and**  $TT': T' \sim_m state\ T$   
**shows**  $full1\ R\ S'\ T' \longleftrightarrow full1\ R-abs\ S\ T$  (**is**  $?R \longleftrightarrow ?R-abs$ )  
 $\langle proof \rangle$

**lemma** *full-if-full-abs*:

**assumes**  $inv\ (state\ S)$  **and**  $full\ R-abs\ S\ T$   
**shows**  $full\ R\ (state\ S)\ (state\ T)$   
 $\langle proof \rangle$

The converse does *not* hold, since we cannot prove that  $S = T$  given  $state\ S = state\ S$ .

**lemma** *full-abs-if-full*:

**assumes**  $inv\ (state\ S)$  **and**  $full\ R\ (state\ S)\ (state\ T)$   
**shows**  $full\ R-abs\ S\ T \vee (state\ S \sim_m state\ T \wedge no-step\ R\ (state\ S))$   
 $\langle proof \rangle$

**lemma** *full-exists-full-abs*:

**assumes**  $inv: inv\ (state\ S)$  **and**  $full: full\ R\ (state\ S)\ T$   
**obtains**  $U$  **where**  $full\ R-abs\ S\ U$  **and**  $T \sim_m state\ U$

$\langle proof \rangle$

**lemma** *full1-exists-full1-abs*:

**assumes** *inv*: *inv* (state *S*) **and** *full1*: *full1* *R* (state *S*) *T*  
**obtains** *U* **where** *full1* *R*-abs *S* *U* **and** *T*  $\sim_m$  state *U*

$\langle proof \rangle$

**lemma** *full1-right-compatible*:

**assumes** *inv* (state *S*) **and**  
*full1*: *full1* *R*-abs *S* *T* **and** *TV*: state *T*  $\sim_m$  state *V*  
**shows** *full1* *R*-abs *S* *V*

$\langle proof \rangle$

**lemma** *full-right-compatible*:

**assumes** *inv*: *inv* (state *S*) **and**  
*full-ST*: *full* *R*-abs *S* *T* **and** *TU*: state *T*  $\sim_m$  state *U*  
**shows** *full* *R*-abs *S* *U*  $\vee$  (*S* = *T*  $\wedge$  no-step *R*-abs *S*)

$\langle proof \rangle$

**end**

**locale** *relation-relation-abs* =

**fixes**

*R* :: '*v* *cdcl<sub>W</sub>*-mset  $\Rightarrow$  '*v* *cdcl<sub>W</sub>*-mset  $\Rightarrow$  bool **and**  
*R*-abs :: '*st*  $\Rightarrow$  '*st*  $\Rightarrow$  bool **and**  
*state* :: '*st*  $\Rightarrow$  '*v* *cdcl<sub>W</sub>*-mset **and**  
*inv* :: '*v* *cdcl<sub>W</sub>*-mset  $\Rightarrow$  bool

**assumes**

*relation-compatible-state*:

*inv* (state *S*)  $\Longrightarrow$  *R* (state *S*) (state *T*)  $\longleftrightarrow$  *R*-abs *S* *T* **and**

*relation-compatible-abs*:

$\bigwedge S S' T. \text{inv } S \Longrightarrow S \sim_m \text{state } S' \Longrightarrow R S T \Longrightarrow \exists U. R\text{-abs } S' U \wedge T \sim_m \text{state } U$  **and**

*relation-invariant*:

$\bigwedge S T. R S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$

**begin**

**lemma** *relation-compatible-eq*:

*inv* (state *S*)  $\Longrightarrow$  *R*-abs *S* *T*  $\Longrightarrow$  state *S*  $\sim_m$  state *S'*  $\Longrightarrow$  state *T*  $\sim_m$  state *T'*  $\Longrightarrow$  *R*-abs *S'* *T'*  
 $\langle proof \rangle$

**lemma** *relation-right-compatible*:

*inv* (state *S*)  $\Longrightarrow$  *R*-abs *S* *T*  $\Longrightarrow$  state *T*  $\sim_m$  state *U*  $\Longrightarrow$  *R*-abs *S* *U*  
 $\langle proof \rangle$

**sublocale** *relation-implied-relation-abs*

$\langle proof \rangle$

**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-statew-ops =
  raw-clss mset-cls
    mset-clss union-clss in-clss insert-clss remove-from-clss
  +
  raw-ccls-union mset-ccls union-ccls remove-clit
for
  — Clause
  mset-cls :: 'cls ⇒ 'v clause and

  — Multiset of Clauses
  mset-clss :: 'clss ⇒ 'v clauses and
  union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
  in-clss :: 'cls ⇒ 'clss ⇒ bool and
  insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and

  mset-ccls :: 'ccls ⇒ 'v clause and
  union-ccls :: 'ccls ⇒ 'ccls ⇒ 'ccls and
  remove-clit :: 'v literal ⇒ 'ccls ⇒ 'ccls
  +
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-conc-init-clss :: 'st ⇒ 'clss and
  raw-conc-learned-clss :: 'st ⇒ 'clss and
  conc-backtrack-lvl :: 'st ⇒ nat and
  raw-conc-conflicting :: 'st ⇒ 'ccls option and

  cons-conc-trail :: ('v, 'cls) ann-lit ⇒ 'st ⇒ 'st and
  tl-conc-trail :: 'st ⇒ 'st and
  add-conc-learned-cls :: 'cls ⇒ 'st ⇒ 'st and
  remove-cls :: 'cls ⇒ 'st ⇒ 'st and
  update-conc-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
  update-conc-conflicting :: 'ccls option ⇒ 'st ⇒ 'st and
  reduce-conc-trail-to :: ('v, 'v clause) ann-lits ⇒ '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]*:  
 $\text{lit-of } ' \text{ mmset-of-mlit } ' \text{ set } M' = \text{lits-of-l } M'$   
 $\langle \text{proof} \rangle$

**lemma** *map-mmset-of-mlit-true-annots-true-cl[simp]*:  
 $\text{map mmset-of-mlit } M' \models_{\text{as}} C \longleftrightarrow M' \models_{\text{as}} C$   
 $\langle \text{proof} \rangle$

**abbreviation** *conc-init-clss*  $\equiv \lambda S. \text{mset-clss } (\text{raw-conc-init-clss } S)$

**abbreviation** *conc-learned-clss*  $\equiv \lambda S. \text{mset-clss } (\text{raw-conc-learned-clss } S)$

**abbreviation** *conc-conflicting*  $\equiv \lambda S. \text{map-option mset-ccls } (\text{raw-conc-conflicting } S)$

**notation** *in-clss* (**infix**  $!\in!$  50)

**notation** *union-clss* (**infix**  $\oplus$  50)

**notation** *insert-clss* (**infix**  $!++!$  50)

**notation** *union-ccls* (**infix**  $!\cup$  50)

**definition** *raw-clauses*  $:: 'st \Rightarrow 'clss$  **where**

*raw-clauses*  $S = \text{union-clss } (\text{raw-conc-init-clss } S) (\text{raw-conc-learned-clss } S)$

**abbreviation** *conc-clauses*  $:: 'st \Rightarrow 'v \text{ clauses}$  **where**

*conc-clauses*  $S \equiv \text{mset-clss } (\text{raw-clauses } S)$

**abbreviation** *resolve-cl* **where**

*resolve-cl*  $L D' E \equiv \text{union-ccls } (\text{remove-clit } (-L) D') (\text{remove-clit } L (\text{ccls-of-clss } E))$

**definition** *state*  $:: 'st \Rightarrow 'v \text{ cdcl}_W\text{-mset}$  **where**

*state*  $= (\lambda S. (\text{conc-trail } S, \text{conc-init-clss } S, \text{conc-learned-clss } S, \text{conc-backtrack-lvl } S, \text{conc-conflicting } S))$

**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.

```

locale abs-stateW =
  abs-stateW-ops
  — functions for clauses:
  mset-cls
    mset-clss union-clss in-clss insert-clss remove-from-clss

  — functions for the conflicting clause:
  mset-ccls union-ccls remove-clit

  — Conversion between conflicting and non-conflicting
  ccls-of-cls cls-of-ccls

  — functions about the state:
    — getter:
    conc-trail hd-raw-conc-trail raw-conc-init-clss raw-conc-learned-clss conc-backtrack-lvl
    raw-conc-conflicting
    — setter:
    cons-conc-trail tl-conc-trail add-conc-learned-cls remove-cls update-conc-backtrack-lvl
    update-conc-conflicting reduce-conc-trail-to

    — Some specific states:
    conc-init-state
    restart-state
for
  mset-cls :: 'cls ⇒ 'v clause and

  mset-clss :: 'clss ⇒ 'v clauses and
  union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
  in-clss :: 'cls ⇒ 'clss ⇒ bool and
  insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and

  mset-ccls :: 'ccls ⇒ 'v clause and
  union-ccls :: 'ccls ⇒ 'ccls ⇒ 'ccls and
  remove-clit :: 'v literal ⇒ 'ccls ⇒ 'ccls 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-conc-init-clss :: 'st ⇒ 'clss and
  raw-conc-learned-clss :: 'st ⇒ 'clss and
  conc-backtrack-lvl :: 'st ⇒ nat and
  raw-conc-conflicting :: 'st ⇒ 'ccls option and

  cons-conc-trail :: ('v, 'cls) ann-lit ⇒ 'st ⇒ 'st and
  tl-conc-trail :: 'st ⇒ 'st and
  add-conc-learned-cls :: 'cls ⇒ 'st ⇒ 'st and
  remove-cls :: 'cls ⇒ 'st ⇒ 'st and
  update-conc-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
  update-conc-conflicting :: 'ccls option ⇒ 'st ⇒ 'st and
  reduce-conc-trail-to :: ('v, 'v clause) ann-lits ⇒ 'st ⇒ 'st and

```

*conc-init-state* :: 'clss  $\Rightarrow$  'st **and**

*restart-state* :: 'st  $\Rightarrow$  'st +

**assumes**

— Definition of *hd-raw-trail*:

*hd-raw-conc-trail*:

*conc-trail*  $S \neq [] \implies \text{mmset-of-mlit } (\text{hd-raw-conc-trail } S) = \text{hd } (\text{conc-trail } S) \text{ and}$

*cons-conc-trail*:

$\bigwedge S'. \text{undefined-lit } (\text{conc-trail } st) \text{ (lit-of } L) \implies$

$\text{state } st = (M, S') \implies$

$\text{state } (\text{cons-conc-trail } L \text{ } st) = (\text{mmset-of-mlit } L \# M, S') \text{ and}$

*tl-conc-trail*:

$\bigwedge S'. \text{state } st = (M, S') \implies \text{state } (\text{tl-conc-trail } st) = (\text{tl } M, S') \text{ and}$

*remove-clss*:

$\bigwedge S'. \text{state } st = (M, N, U, S') \implies$

$\text{state } (\text{remove-clss } C \text{ } st) =$

$(M, \text{removeAll-mset } (\text{mset-clss } C) \text{ } N, \text{removeAll-mset } (\text{mset-clss } C) \text{ } U, S') \text{ and}$

*add-conc-learned-clss*:

$\bigwedge S'. \text{no-dup } (\text{conc-trail } st) \implies \text{state } st = (M, N, U, S') \implies$

$\text{state } (\text{add-conc-learned-clss } C \text{ } st) =$

$(M, N, \{\# \text{mset-clss } C \# \} + U, S') \text{ and}$

*update-conc-backtrack-lvl*:

$\bigwedge S'. \text{state } st = (M, N, U, k, S') \implies$

$\text{state } (\text{update-conc-backtrack-lvl } k' \text{ } st) = (M, N, U, k', S') \text{ and}$

*update-conc-conflicting*:

$\text{state } st = (M, N, U, k, D) \implies$

$\text{state } (\text{update-conc-conflicting } E \text{ } st) = (M, N, U, k, \text{map-option mset-clss } E) \text{ and}$

*conc-conflicting-update-conc-conflicting[simp]*:

$\text{raw-conc-conflicting } (\text{update-conc-conflicting } E \text{ } st) = E \text{ and}$

*conc-init-state*:

$\text{state } (\text{conc-init-state } Ns) = ([], \text{mset-clss } Ns, \{\#\}, 0, \text{None}) \text{ and}$

— Properties about restarting *restart-state*:

*conc-trail-restart-state[simp]*:  $\text{conc-trail } (\text{restart-state } S) = [] \text{ and}$

*conc-init-clss-restart-state[simp]*:  $\text{conc-init-clss } (\text{restart-state } S) = \text{conc-init-clss } S \text{ and}$

*conc-learned-clss-restart-state[intro]*:

$\text{conc-learned-clss } (\text{restart-state } S) \subseteq \# \text{conc-learned-clss } S \text{ and}$

*conc-backtrack-lvl-restart-state[simp]*:  $\text{conc-backtrack-lvl } (\text{restart-state } S) = 0 \text{ and}$

*conc-conflicting-restart-state[simp]*:  $\text{conc-conflicting } (\text{restart-state } S) = \text{None} \text{ and}$

— Properties about *reduce-conc-trail-to*:

*reduce-conc-trail-to[simp]*:

$\bigwedge S'. \text{conc-trail } st = M2 @ M1 \implies \text{state } st = (M, S') \implies$

$\text{state } (\text{reduce-conc-trail-to } M1 \text{ } st) = (M1, S')$

**begin**

**lemma**

— Properties about the trail *conc-trail*:

*conc-trail-cons-conc-trail*[simp]:  
 $\text{undefined-lit } (\text{conc-trail } st) (\text{lit-of } L) \implies$   
 $\text{conc-trail } (\text{cons-conc-trail } L \ st) = \text{mmset-of-mlit } L \ \# \ \text{conc-trail } st \text{ and}$   
*conc-trail-tl-conc-trail*[simp]:  
 $\text{conc-trail } (\text{tl-conc-trail } st) = \text{tl } (\text{conc-trail } st) \text{ and}$   
*conc-trail-add-conc-learned-cls*[simp]:  
 $\text{no-dup } (\text{conc-trail } st) \implies \text{conc-trail } (\text{add-conc-learned-cls } C \ st) = \text{conc-trail } st \text{ and}$   
*conc-trail-remove-cls*[simp]:  
 $\text{conc-trail } (\text{remove-cls } C \ st) = \text{conc-trail } st \text{ and}$   
*conc-trail-update-conc-backtrack-lvl*[simp]:  
 $\text{conc-trail } (\text{update-conc-backtrack-lvl } k \ st) = \text{conc-trail } st \text{ and}$   
*conc-trail-update-conc-conflicting*[simp]:  
 $\text{conc-trail } (\text{update-conc-conflicting } E \ st) = \text{conc-trail } st \text{ and}$

— Properties about the initial clauses *conc-init-clss*:

*conc-init-clss-cons-conc-trail*[simp]:  
 $\text{undefined-lit } (\text{conc-trail } st) (\text{lit-of } L) \implies$   
 $\text{conc-init-clss } (\text{cons-conc-trail } L \ st) = \text{conc-init-clss } st$   
**and**  
*conc-init-clss-tl-conc-trail*[simp]:  
 $\text{conc-init-clss } (\text{tl-conc-trail } st) = \text{conc-init-clss } st \text{ and}$   
*conc-init-clss-add-conc-learned-cls*[simp]:  
 $\text{no-dup } (\text{conc-trail } st) \implies$   
 $\text{conc-init-clss } (\text{add-conc-learned-cls } C \ st) = \text{conc-init-clss } st \text{ and}$   
*conc-init-clss-remove-cls*[simp]:  
 $\text{conc-init-clss } (\text{remove-cls } C \ st) = \text{removeAll-mset } (\text{mset-cls } C) (\text{conc-init-clss } st) \text{ and}$   
*conc-init-clss-update-conc-backtrack-lvl*[simp]:  
 $\text{conc-init-clss } (\text{update-conc-backtrack-lvl } k \ st) = \text{conc-init-clss } st \text{ and}$   
*conc-init-clss-update-conc-conflicting*[simp]:  
 $\text{conc-init-clss } (\text{update-conc-conflicting } E \ st) = \text{conc-init-clss } st \text{ and}$

— Properties about the learned clauses *conc-learned-clss*:

*conc-learned-clss-cons-conc-trail*[simp]:  
 $\text{undefined-lit } (\text{conc-trail } st) (\text{lit-of } L) \implies$   
 $\text{conc-learned-clss } (\text{cons-conc-trail } L \ st) = \text{conc-learned-clss } st \text{ and}$   
*conc-learned-clss-tl-conc-trail*[simp]:  
 $\text{conc-learned-clss } (\text{tl-conc-trail } st) = \text{conc-learned-clss } st \text{ and}$   
*conc-learned-clss-add-conc-learned-cls*[simp]:  
 $\text{no-dup } (\text{conc-trail } st) \implies$   
 $\text{conc-learned-clss } (\text{add-conc-learned-cls } C \ st) = \{\# \text{mset-cls } C \# \} + \text{conc-learned-clss } st \text{ and}$   
*conc-learned-clss-remove-cls*[simp]:  
 $\text{conc-learned-clss } (\text{remove-cls } C \ st) = \text{removeAll-mset } (\text{mset-cls } C) (\text{conc-learned-clss } st) \text{ and}$   
*conc-learned-clss-update-conc-backtrack-lvl*[simp]:  
 $\text{conc-learned-clss } (\text{update-conc-backtrack-lvl } k \ st) = \text{conc-learned-clss } st \text{ and}$   
*conc-learned-clss-update-conc-conflicting*[simp]:  
 $\text{conc-learned-clss } (\text{update-conc-conflicting } E \ st) = \text{conc-learned-clss } st \text{ and}$

— Properties about the backtracking level *conc-backtrack-lvl*:

*conc-backtrack-lvl-cons-conc-trail*[simp]:  
 $\text{undefined-lit } (\text{conc-trail } st) (\text{lit-of } L) \implies$   
 $\text{conc-backtrack-lvl } (\text{cons-conc-trail } L \ st) = \text{conc-backtrack-lvl } st \text{ and}$   
*conc-backtrack-lvl-tl-conc-trail*[simp]:  
 $\text{conc-backtrack-lvl } (\text{tl-conc-trail } st) = \text{conc-backtrack-lvl } st \text{ and}$   
*conc-backtrack-lvl-add-conc-learned-cls*[simp]:  
 $\text{no-dup } (\text{conc-trail } st) \implies$   
 $\text{conc-backtrack-lvl } (\text{add-conc-learned-cls } C \ st) = \text{conc-backtrack-lvl } st \text{ and}$

*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-update-conc-conflicting*[simp]:  
*conc-backtrack-lvl* (update-conc-conflicting  $E$   $st$ ) = *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  $L$ )  $\implies$   
*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-learned-cls*[simp]:  
 no-dup (*conc-trail*  $st$ )  $\implies$   
*conc-conflicting* (add-conc-learned-cls  $C$   $st$ ) = *conc-conflicting*  $st$   
**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**

— 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*  $st$  =  $M2$  @  $M1$   $\implies$  *conc-trail* (*reduce-conc-trail-to*  $M1$   $st$ ) =  $M1$  **and**  
*conc-init-clss-reduce-conc-trail-to*[simp]:  
*conc-trail*  $st$  =  $M2$  @  $M1$   $\implies$   
*conc-init-clss* (*reduce-conc-trail-to*  $M1$   $st$ ) = *conc-init-clss*  $st$  **and**  
*conc-learned-clss-reduce-conc-trail-to*[simp]:  
*conc-trail*  $st$  =  $M2$  @  $M1$   $\implies$   
*conc-learned-clss* (*reduce-conc-trail-to*  $M1$   $st$ ) = *conc-learned-clss*  $st$  **and**  
*conc-backtrack-lvl-reduce-conc-trail-to*[simp]:  
*conc-trail*  $st$  =  $M2$  @  $M1$   $\implies$   
*conc-backtrack-lvl* (*reduce-conc-trail-to*  $M1$   $st$ ) = *conc-backtrack-lvl*  $st$  **and**  
*conc-conflicting-reduce-conc-trail-to*[simp]:  
*conc-trail*  $st$  =  $M2$  @  $M1$   $\implies$   
*conc-conflicting* (*reduce-conc-trail-to*  $M1$   $st$ ) = *conc-conflicting*  $st$   
 <proof>

**lemma**

**shows**

*clauses-cons-conc-trail*[simp]:  
 undefined-lit (*conc-trail*  $S$ ) (lit-of  $L$ )  $\implies$   
*conc-clauses* (cons-conc-trail  $L$   $S$ ) = *conc-clauses*  $S$  **and**  
*clss-tl-conc-trail*[simp]: *conc-clauses* (tl-conc-trail  $S$ ) = *conc-clauses*  $S$  **and**  
*clauses-add-conc-learned-cls-unfolded*:  
 no-dup (*conc-trail*  $S$ )  $\implies$  *conc-clauses* (add-conc-learned-cls  $U$   $S$ ) =



$\{\#mset-cl\ S\} + conc-learned-clss\ S + conc-init-clss\ S$   
**and**  
*clauses-update-conc-backtrack-lvl*[simp]:  
 $conc-clauses\ (update-conc-backtrack-lvl\ k\ S) = conc-clauses\ S$  **and**  
*clauses-update-conc-conflicting*[simp]:  
 $conc-clauses\ (update-conc-conflicting\ D\ S) = conc-clauses\ S$  **and**  
*clauses-remove-cl*[simp]:  
 $conc-clauses\ (remove-cl\ C\ S) = removeAll-mset\ (mset-cl\ C)\ (conc-clauses\ S)$  **and**  
*clauses-add-conc-learned-cl*[simp]:  
 $no-dup\ (conc-trail\ S) \implies$   
 $conc-clauses\ (add-conc-learned-cl\ C\ S) = \{\#mset-cl\ C\} + conc-clauses\ S$  **and**  
*clauses-restart*[simp]:  $conc-clauses\ (restart-state\ S) \subseteq \# conc-clauses\ S$  **and**  
*clauses-conc-init-state*[simp]:  $\bigwedge N. conc-clauses\ (conc-init-state\ N) = mset-clss\ N$   
 $\langle proof \rangle$

**abbreviation** *incr-lvl* ::  $'st \Rightarrow 'st$  **where**

$incr-lvl\ S \equiv update-conc-backtrack-lvl\ (conc-backtrack-lvl\ S + 1)\ S$

**abbreviation** *state-eq* ::  $'st \Rightarrow 'st \Rightarrow bool$  (**infix**  $\sim 36$ ) **where**

$S \sim T \equiv state\ S \sim_m state\ T$

**lemma** *state-eq-sym*:

$S \sim T \longleftrightarrow T \sim S$

$\langle proof \rangle$

**lemma** *state-eq-trans*:

$S \sim T \implies T \sim U \implies S \sim U$

$\langle proof \rangle$

**lemma**

**shows**

*state-eq-conc-trail*:  $S \sim T \implies conc-trail\ S = conc-trail\ T$  **and**

*state-eq-conc-init-clss*:  $S \sim T \implies conc-init-clss\ S = conc-init-clss\ T$  **and**

*state-eq-conc-learned-clss*:  $S \sim T \implies conc-learned-clss\ S = conc-learned-clss\ T$  **and**

*state-eq-conc-backtrack-lvl*:  $S \sim T \implies conc-backtrack-lvl\ S = conc-backtrack-lvl\ T$  **and**

*state-eq-conc-conflicting*:  $S \sim T \implies conc-conflicting\ S = conc-conflicting\ T$  **and**

*state-eq-clauses*:  $S \sim T \implies conc-clauses\ S = conc-clauses\ T$  **and**

*state-eq-undefined-lit*:

$S \sim T \implies undefined-lit\ (conc-trail\ S)\ L = undefined-lit\ (conc-trail\ T)\ L$

$\langle proof \rangle$

We combine all simplification rules about  $op \sim$  in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

**lemmas** *state-simp* = *state-eq-conc-trail* *state-eq-conc-init-clss* *state-eq-conc-learned-clss*

*state-eq-conc-backtrack-lvl* *state-eq-conc-conflicting* *state-eq-clauses* *state-eq-undefined-lit*

**lemma** *atms-of-ms-conc-learned-clss-restart-state-in-atms-of-ms-conc-learned-clssI*[intro]:

$x \in atms-of-mm\ (conc-learned-clss\ (restart-state\ S)) \implies x \in atms-of-mm\ (conc-learned-clss\ S)$

$\langle proof \rangle$

**lemma** *clauses-reduce-conc-trail-to*[simp]:

$conc-trail\ S = M2\ @\ M1 \implies conc-clauses\ (reduce-conc-trail-to\ M1\ S) = conc-clauses\ S$

$\langle proof \rangle$

**lemma** *in-get-all-ann-decomposition-conc-trail-update-conc-trail[simp]*:  
**assumes**  $H: (L \# M1, M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{conc-trail } S))$   
**shows**  $\text{conc-trail } (\text{reduce-conc-trail-to } M1 \ S) = M1$   
 $\langle \text{proof} \rangle$

**lemma** *raw-conc-conflicting-cons-conc-trail[simp]*:  
**assumes**  $\text{undefined-lit } (\text{conc-trail } S) \ (\text{lit-of } L)$   
**shows**  
 $\text{raw-conc-conflicting } (\text{cons-conc-trail } L \ S) = \text{None} \longleftrightarrow \text{raw-conc-conflicting } S = \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *raw-conc-conflicting-add-conc-learned-cl[simp]*:  
 $\text{no-dup } (\text{conc-trail } S) \implies$   
 $\text{raw-conc-conflicting } (\text{add-conc-learned-cl } C \ S) = \text{None} \longleftrightarrow \text{raw-conc-conflicting } S = \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *raw-conc-conflicting-update-backtrack-lvl[simp]*:  
 $\text{raw-conc-conflicting } (\text{update-conc-backtrack-lvl } k \ S) = \text{None} \longleftrightarrow \text{raw-conc-conflicting } S = \text{None}$   
 $\langle \text{proof} \rangle$

**end** — end of  $\text{state}_W$  locale

## 7.2.4 CDCL Rules

**locale** *abs-conflict-driven-clause-learning<sub>W</sub>* =  
 $\text{abs-state}_W$   
— functions for clauses:  
 $\text{mset-cl}$   
 $\text{mset-clss union-clss in-clss insert-clss remove-from-clss}$   
— functions for the conflicting clause:  
 $\text{mset-ccls union-ccls remove-clit}$   
— conversion  
 $\text{ccls-of-cl } \text{cl-of-ccls}$   
— functions for the state:  
— access functions:  
 $\text{conc-trail } \text{hd-raw-conc-trail } \text{raw-conc-init-clss } \text{raw-conc-learned-clss } \text{conc-backtrack-lvl}$   
 $\text{raw-conc-conflicting}$   
— changing state:  
 $\text{cons-conc-trail } \text{tl-conc-trail } \text{add-conc-learned-cl } \text{remove-cl } \text{update-conc-backtrack-lvl}$   
 $\text{update-conc-conflicting } \text{reduce-conc-trail-to}$   
— get state:  
 $\text{conc-init-state}$   
 $\text{restart-state}$   
**for**  
 $\text{mset-cl} :: 'cl \Rightarrow 'v \text{ clause}$  **and**  
 $\text{mset-clss} :: 'clss \Rightarrow 'v \text{ clauses}$  **and**  
 $\text{union-clss} :: 'clss \Rightarrow 'clss \Rightarrow 'clss$  **and**  
 $\text{in-clss} :: 'cl \Rightarrow 'clss \Rightarrow \text{bool}$  **and**  
 $\text{insert-clss} :: 'cl \Rightarrow 'clss \Rightarrow 'clss$  **and**  
 $\text{remove-from-clss} :: 'cl \Rightarrow 'clss \Rightarrow 'clss$  **and**

*mset-ccls* :: 'ccls  $\Rightarrow$  'v clause **and**  
*union-ccls* :: 'ccls  $\Rightarrow$  'ccls  $\Rightarrow$  'ccls **and**  
*remove-clit* :: 'v literal  $\Rightarrow$  'ccls  $\Rightarrow$  'ccls **and**

*ccls-of-cl* :: 'cls  $\Rightarrow$  'ccls **and**  
*cls-of-ccls* :: 'ccls  $\Rightarrow$  'cls **and**

*conc-trail* :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits **and**  
*hd-raw-conc-trail* :: 'st  $\Rightarrow$  ('v, 'cls) ann-lit **and**  
*raw-conc-init-clss* :: 'st  $\Rightarrow$  'clss **and**  
*raw-conc-learned-clss* :: 'st  $\Rightarrow$  'clss **and**  
*conc-backtrack-lvl* :: 'st  $\Rightarrow$  nat **and**  
*raw-conc-conflicting* :: 'st  $\Rightarrow$  'ccls option **and**

*cons-conc-trail* :: ('v, 'cls) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*tl-conc-trail* :: 'st  $\Rightarrow$  'st **and**  
*add-conc-learned-cl* :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*remove-cl* :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*update-conc-backtrack-lvl* :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*update-conc-conflicting* :: 'ccls option  $\Rightarrow$  'st  $\Rightarrow$  'st **and**  
*reduce-conc-trail-to* :: ('v, 'v clause) ann-lits  $\Rightarrow$  'st  $\Rightarrow$  'st **and**

*conc-init-state* :: 'clss  $\Rightarrow$  'st **and**  
*restart-state* :: 'st  $\Rightarrow$  'st

**begin**

**lemma** *clauses-state-conc-clauses[simp]*:  $\text{cdcl}_W\text{-mset.clauses (state } S) = \text{conc-clauses } S$   
 <proof>

**lemma** *conflicting-None-iff-raw-conc-conflicting[simp]*:  
 $\text{conflicting (state } S) = \text{None} \longleftrightarrow \text{raw-conc-conflicting } S = \text{None}$   
 <proof>

**lemma** *trail-state-add-conc-learned-cl*:  
 $\text{no-dup (conc-trail } S) \implies \text{trail (state (add-conc-learned-cl } D \text{ } S)) = \text{trail (state } S)$   
 <proof>

**lemma** *trail-state-update-backtrack-lvl*:  
 $\text{trail (state (update-conc-backtrack-lvl } i \text{ } S)) = \text{trail (state } S)$   
 <proof>

**lemma** *trail-state-update-conflicting*:  
 $\text{trail (state (update-conc-conflicting } i \text{ } S)) = \text{trail (state } S)$   
 <proof>

**lemma** *trail-state-conc-trail[simp]*:  
 $\text{trail (state } S) = \text{conc-trail } S$   
 <proof>

**lemma** *init-clss-state-conc-init-clss[simp]*:  
 $\text{init-clss (state } S) = \text{conc-init-clss } S$   
 <proof>

**lemma** *learned-clss-state-conc-learned-clss[simp]*:  
 $\text{learned-clss (state } S) = \text{conc-learned-clss } S$   
 <proof>

**lemma** *tl-trail-state-tl-con-trail*[simp]:  
 $tl-trail\ (state\ S) = state\ (tl-conc-trail\ S)$   
 $\langle proof \rangle$

**lemma** *add-learned-cls-state-add-conc-learned-cls*[simp]:  
**assumes** *no-dup* (*conc-trail* *S*)  
**shows** *add-learned-cls* (*mset-ccls* *D'*) (*state* *S*) = *state* (*add-conc-learned-cls* (*cls-of-ccls* *D'*) *S*)  
 $\langle proof \rangle$

**lemma** *state-cons-cons-trail-cons-trail*[simp]:  
 $undefined-lit\ (trail\ (state\ S))\ (lit-of\ L) \implies$   
 $cons-trail\ (mset-of-mlit\ L)\ (state\ S) = state\ (cons-conc-trail\ L\ S)$   
 $\langle proof \rangle$

**lemma** *state-cons-cons-trail-cons-trail-propagated*[simp]:  
 $undefined-lit\ (trail\ (state\ S))\ K \implies$   
 $cons-trail\ (Propagated\ K\ (mset-cls\ C))\ (state\ S) = state\ (cons-conc-trail\ (Propagated\ K\ C)\ S)$   
 $\langle proof \rangle$

**lemma** *state-cons-cons-trail-cons-trail-propagated-ccls*[simp]:  
 $undefined-lit\ (trail\ (state\ S))\ K \implies$   
 $cons-trail\ (Propagated\ K\ (mset-ccls\ C))\ (state\ S) =$   
 $state\ (cons-conc-trail\ (Propagated\ K\ (cls-of-ccls\ C))\ S)$   
 $\langle proof \rangle$

**lemma** *state-cons-cons-trail-cons-trail-decided*[simp]:  
 $undefined-lit\ (trail\ (state\ S))\ K \implies$   
 $cons-trail\ (Decided\ K)\ (state\ S) = state\ (cons-conc-trail\ (Decided\ K)\ S)$   
 $\langle proof \rangle$

**lemma** *state-update-conc-conflicting-update-conflicting*[simp]:  
 $update-conflicting\ (Some\ (mset-ccls\ D))\ (state\ S) = state\ (update-conc-conflicting\ (Some\ D)\ S)$   
 $update-conflicting\ (Some\ (mset-cls\ D'))\ (state\ S) =$   
 $state\ (update-conc-conflicting\ (Some\ (ccls-of-cls\ D'))\ S)$   
 $\langle proof \rangle$

**lemma** *update-conflicting-None-state*[simp]:  
 $update-conflicting\ None\ (state\ S) = state\ (update-conc-conflicting\ None\ S)$   
 $\langle proof \rangle$

**lemma** *update-backtrack-lvl-state*[simp]:  
 $update-backtrack-lvl\ i\ (state\ S) = state\ (update-conc-backtrack-lvl\ i\ S)$   
 $\langle proof \rangle$

**lemma** *conc-conflicting-conflicting*[simp]:  
 $conflicting\ (state\ S) = conc-conflicting\ S$   
 $\langle proof \rangle$

**lemma** *update-conflicting-resolve-state-update-conc-conflicting*[simp]:  
 $update-conflicting\ (Some\ (remove1-mset\ (-\ L)\ (mset-ccls\ D')\ \# \cup\ remove1-mset\ L\ (mset-cls\ E')))$   
 $(state\ (tl-conc-trail\ S)) =$   
 $state\ (update-conc-conflicting\ (Some\ (resolve-cls\ L\ D'\ E'))\ (tl-conc-trail\ S))$   
 $\langle proof \rangle$

**lemma** *conc-backtrack-lvl-backtrack-lvl*[simp]:

*backtrack-lvl* (state *S*) = *conc-backtrack-lvl* *S*  
 ⟨proof⟩

**lemma** *state-state*:

*cdcl<sub>W</sub>-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<sub>W</sub>-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<sub>W</sub>-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<sub>W</sub>-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* ⊨<sub>as</sub> 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<sub>W</sub>-mset.propagate* (state *S*) (state *T*) ⟷ *propagate-abs* *S* *T* (is ?*mset* ⟷ ?*abs*)  
 ⟨proof⟩

**lemma** *propagate-compatible-abs*:

**assumes** *SS'*: *S* ∼<sub>m</sub> state *S'* **and** *abs*: *cdcl<sub>W</sub>-mset.propagate* *S* *T*

**obtains** *U* **where** *propagate-abs* *S'* *U* **and** *T* ∼<sub>m</sub> state *U*

⟨proof⟩

**interpretation** *propagate-abs*: *relation-relation-abs* *cdcl<sub>W</sub>-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* ⊨<sub>as</sub> CNot (*mset-cls* *D*) ⇒

$T \sim \text{update-conc-conflicting } (\text{Some } (\text{ccls-of-cls } D)) \ S \implies$   
 $\text{conflict-abs } S \ T$

**inductive-cases**  $\text{conflict-absE}$ :  $\text{conflict-abs } S \ T$

**lemma**  $\text{conflict-conflict-abs}$ :

$\text{cdcl}_W\text{-mset.conflict } (\text{state } S) \ (\text{state } T) \longleftrightarrow \text{conflict-abs } S \ T \ (\text{is } ?\text{mset} \longleftrightarrow ?\text{abs})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{conflict-compatible-abs}$ :

**assumes**  $SS'$ :  $S \sim_m \text{state } S'$  **and**  $\text{conflict}$ :  $\text{cdcl}_W\text{-mset.conflict } S \ T$   
**obtains**  $U$  **where**  $\text{conflict-abs } S' \ U$  **and**  $T \sim_m \text{state } U$   
 $\langle \text{proof} \rangle$

**interpretation**  $\text{conflict-abs}$ :  $\text{relation-relation-abs } \text{cdcl}_W\text{-mset.conflict } \text{conflict-abs } \text{state}$

$\lambda\cdot. \text{True}$

$\langle \text{proof} \rangle$

**inductive**  $\text{backtrack-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

$\text{backtrack-abs-rule}$ :

$\text{raw-conc-conflicting } S = \text{Some } D \implies$   
 $L \in \# \text{ mset-ccls } D \implies$   
 $(\text{Decided } K \ \# \ M1, \ M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{conc-trail } S)) \implies$   
 $\text{get-level } (\text{conc-trail } S) \ L = \text{conc-backtrack-lvl } S \implies$   
 $\text{get-level } (\text{conc-trail } S) \ L = \text{get-maximum-level } (\text{conc-trail } S) \ (\text{mset-ccls } D) \implies$   
 $\text{get-maximum-level } (\text{conc-trail } S) \ (\text{mset-ccls } D - \{\#L\# \}) \equiv i \implies$   
 $\text{get-level } (\text{conc-trail } S) \ K = i + 1 \implies$   
 $T \sim \text{cons-conc-trail } (\text{Propagated } L \ (\text{cls-of-ccls } D))$   
 $(\text{reduce-conc-trail-to } M1$   
 $(\text{add-conc-learned-cls } (\text{cls-of-ccls } D)$   
 $(\text{update-conc-backtrack-lvl } i$   
 $(\text{update-conc-conflicting } \text{None } S)))) \implies$   
 $\text{backtrack-abs } S \ T$

**inductive-cases**  $\text{backtrack-absE}$ :  $\text{backtrack-abs } S \ T$

**lemma**  $\text{backtrack-backtrack-abs}$ :

**assumes**  $\text{inv}$ :  $\text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv } (\text{state } S)$   
**shows**  $\text{cdcl}_W\text{-mset.backtrack } (\text{state } S) \ (\text{state } T) \longleftrightarrow \text{backtrack-abs } S \ T \ (\text{is } ?\text{conc} \longleftrightarrow ?\text{abs})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{backtrack-exists-backtrack-abs-step}$ :

**assumes**  $\text{bt}$ :  $\text{cdcl}_W\text{-mset.backtrack } S \ T$  **and**  $\text{inv}$ :  $\text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv } S$  **and**  
 $SS'$ :  $S \sim_m \text{state } S'$   
**obtains**  $U$  **where**  $\text{backtrack-abs } S' \ U$  **and**  $T \sim_m \text{state } U$   
 $\langle \text{proof} \rangle$

**interpretation**  $\text{backtrack-abs}$ :  $\text{relation-relation-abs } \text{cdcl}_W\text{-mset.backtrack } \text{backtrack-abs } \text{state}$

$\text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv}$

$\langle \text{proof} \rangle$

**inductive**  $\text{decide-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

$\text{decide-abs-rule}$ :

$\text{conc-conflicting } S = \text{None} \implies$   
 $\text{undefined-lit } (\text{conc-trail } S) \ L \implies$   
 $\text{atm-of } L \in \text{atms-of-mm } (\text{conc-init-cls } S) \implies$

$T \sim \text{cons-conc-trail } (\text{Decided } L) (\text{incr-lvl } S) \implies$   
 $\text{decide-abs } S \ T$

**inductive-cases**  $\text{decide-absE}$ :  $\text{decide-abs } S \ T$

**lemma**  $\text{decide-decide-abs}$ :

$\text{cdcl}_W\text{-mset.decide } (\text{state } S) (\text{state } T) \longleftrightarrow \text{decide-abs } S \ T$   
 $\langle \text{proof} \rangle$

**interpretation**  $\text{decide-abs}$ :  $\text{relation-relation-abs } \text{cdcl}_W\text{-mset.decide } \text{decide-abs state}$

$\lambda\cdot. \text{True}$   
 $\langle \text{proof} \rangle$

**inductive**  $\text{skip-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

$\text{skip-abs-rule}$ :

$\text{conc-trail } S = \text{Propagated } L \ C' \ \# \ M \implies$   
 $\text{raw-conc-conflicting } S = \text{Some } E \implies$   
 $-L \notin \# \ \text{mset-ccls } E \implies$   
 $\text{mset-ccls } E \neq \{\#\} \implies$   
 $T \sim \text{tl-conc-trail } S \implies$   
 $\text{skip-abs } S \ T$

**inductive-cases**  $\text{skip-absE}$ :  $\text{skip-abs } S \ T$

**lemma**  $\text{skip-skip-abs}$ :

$\text{cdcl}_W\text{-mset.skip } (\text{state } S) (\text{state } T) \longleftrightarrow \text{skip-abs } S \ T \ (\text{is } ?\text{conc} \longleftrightarrow ?\text{abs})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{skip-exists-skip-abs}$ :

**assumes**  $\text{skip}$ :  $\text{cdcl}_W\text{-mset.skip } S \ T$  **and**  $SS'$ :  $S \sim_m \text{state } S'$

**obtains**  $U$  **where**  $\text{skip-abs } S' \ U$  **and**  $T \sim_m \text{state } U$

$\langle \text{proof} \rangle$

**interpretation**  $\text{skip-abs}$ :  $\text{relation-relation-abs } \text{cdcl}_W\text{-mset.skip } \text{skip-abs state}$

$\lambda\cdot. \text{True}$   
 $\langle \text{proof} \rangle$

**inductive**  $\text{resolve-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

$\text{resolve-abs-rule}$ :  $\text{conc-trail } S \neq [] \implies$

$\text{hd-raw-conc-trail } S = \text{Propagated } L \ E \implies$

$L \in \# \ \text{mset-cls } E \implies$

$\text{raw-conc-conflicting } S = \text{Some } D' \implies$

$-L \in \# \ \text{mset-ccls } D' \implies$

$\text{get-maximum-level } (\text{conc-trail } S) (\text{mset-ccls } (\text{remove-clit } (-L) \ D')) = \text{conc-backtrack-lvl } S \implies$

$T \sim \text{update-conc-conflicting } (\text{Some } (\text{resolve-cls } L \ D' \ E))$

$(\text{tl-conc-trail } S) \implies$

$\text{resolve-abs } S \ T$

**inductive-cases**  $\text{resolve-absE}$ :  $\text{resolve-abs } S \ T$

**lemma**  $\text{resolve-resolve-abs}$ :

$\text{cdcl}_W\text{-mset.resolve } (\text{state } S) (\text{state } T) \longleftrightarrow \text{resolve-abs } S \ T \ (\text{is } ?\text{conc} \longleftrightarrow ?\text{abs})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{resolve-exists-resolve-abs}$ :

**assumes**

**res:**  $cdcl_W\text{-mset.resolve } S \ T$  **and**  
**SS':**  $S \sim_m \text{state } S'$   
**obtains**  $U$  **where**  $\text{resolve-abs } S' \ U$  **and**  $T \sim_m \text{state } U$   
 <proof>

**interpretation**  $\text{resolve-abs}$ :  $\text{relation-relation-abs } cdcl_W\text{-mset.resolve resolve-abs state}$   
 $\lambda\cdot. \text{True}$   
 <proof>

**inductive**  $\text{restart} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**  
 $\text{restart: conc-conflicting } S = \text{None} \implies$   
 $\neg \text{conc-trail } S \models_{\text{asm}} \text{conc-clauses } S \implies$   
 $T \sim \text{restart-state } S \implies$   
 $\text{restart } S \ T$

**inductive-cases**  $\text{restartE}$ :  $\text{restart } S \ T$

We add the condition  $C \notin \# \text{conc-init-clss } S$ , to maintain consistency even without the strategy.

**inductive**  $\text{forget} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **where**  
 $\text{forget-rule:}$

$\text{conc-conflicting } S = \text{None} \implies$   
 $C \notin \# \text{raw-conc-learned-clss } S \implies$   
 $\neg (\text{conc-trail } S) \models_{\text{asm}} \text{clauses } S \implies$   
 $\text{mset-clss } C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{conc-trail } S)) \implies$   
 $\text{mset-clss } C \notin \# \text{conc-init-clss } S \implies$   
 $T \sim \text{remove-clss } C \ S \implies$   
 $\text{forget } S \ T$

**inductive-cases**  $\text{forgetE}$ :  $\text{forget } S \ T$

**inductive**  $cdcl_W\text{-abs-rf} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**  
 $\text{restart: restart-abs } S \ T \implies cdcl_W\text{-abs-rf } S \ T \mid$   
 $\text{forget: forget-abs } S \ T \implies cdcl_W\text{-abs-rf } S \ T$

**inductive**  $cdcl_W\text{-abs-bj} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **where**  
 $\text{skip: skip-abs } S \ S' \implies cdcl_W\text{-abs-bj } S \ S' \mid$   
 $\text{resolve: resolve-abs } S \ S' \implies cdcl_W\text{-abs-bj } S \ S' \mid$   
 $\text{backtrack: backtrack-abs } S \ S' \implies cdcl_W\text{-abs-bj } S \ S'$

**inductive-cases**  $cdcl_W\text{-abs-bjE}$ :  $cdcl_W\text{-abs-bj } S \ T$

**lemma**  $cdcl_W\text{-abs-bj-cdcl}_W\text{-abs-bj}$ :  
 $cdcl_W\text{-mset.cdcl}_W\text{-all-struct-inv } (\text{state } S) \implies$   
 $cdcl_W\text{-mset.cdcl}_W\text{-bj } (\text{state } S) \ (\text{state } T) \longleftrightarrow cdcl_W\text{-abs-bj } S \ T$   
 <proof>

**interpretation**  $cdcl_W\text{-abs-bj}$ :  $\text{relation-relation-abs } cdcl_W\text{-mset.cdcl}_W\text{-bj } cdcl_W\text{-abs-bj state}$   
 $cdcl_W\text{-mset.cdcl}_W\text{-all-struct-inv}$   
 <proof>

**inductive**  $cdcl_W\text{-abs-o} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**  
 $\text{decide: decide-abs } S \ S' \implies cdcl_W\text{-abs-o } S \ S' \mid$   
 $\text{bj: cdcl}_W\text{-abs-bj } S \ S' \implies cdcl_W\text{-abs-o } S \ S'$

**inductive**  $cdcl_W\text{-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**  
 $\text{propagate: propagate-abs } S \ S' \implies cdcl_W\text{-abs } S \ S' \mid$



*conflict*:  $\text{conflict-abs } S \ S' \implies \text{cdcl}_W\text{-abs } S \ S' \mid$   
*other*:  $\text{cdcl}_W\text{-abs-o } S \ S' \implies \text{cdcl}_W\text{-abs } S \ S' \mid$   
*rf*:  $\text{cdcl}_W\text{-abs-rf } S \ S' \implies \text{cdcl}_W\text{-abs } S \ S'$

### 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**  $\text{cdcl}_W\text{-merge-abs-cp} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**  
*conflict'*:  $\text{conflict-abs } S \ T \implies \text{full } \text{cdcl}_W\text{-abs-bj } T \ U \implies \text{cdcl}_W\text{-merge-abs-cp } S \ U \mid$   
*propagate'*:  $\text{propagate-abs}^{++} \ S \ S' \implies \text{cdcl}_W\text{-merge-abs-cp } S \ S'$

**lemma**  $\text{cdcl}_W\text{-merge-cp-cdcl}_W\text{-abs-merge-cp}$ :

**assumes**

$\text{cp} : \text{cdcl}_W\text{-merge-abs-cp } S \ T$  **and**

$\text{inv} : \text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv (state } S)$

**shows**  $\text{cdcl}_W\text{-mset.cdcl}_W\text{-merge-cp (state } S) \text{ (state } T)$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-merge-cp-abs-exists-cdcl}_W\text{-merge-cp}$ :

**assumes**

$\text{cp} : \text{cdcl}_W\text{-mset.cdcl}_W\text{-merge-cp (state } S) \ T$  **and**

$\text{inv} : \text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv (state } S)$

**obtains**  $U$  **where**  $\text{cdcl}_W\text{-merge-abs-cp } S \ U$  **and**  $T \sim_m \text{state } U$

$\langle \text{proof} \rangle$

**lemma**  $\text{no-step-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-abs-merge-cp}$ :

**assumes**

$\text{inv} : \text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv (state } S)$

**shows**  $\text{no-step } \text{cdcl}_W\text{-merge-abs-cp } S \longleftrightarrow \text{no-step } \text{cdcl}_W\text{-mset.cdcl}_W\text{-merge-cp (state } S)$

(**is**  $?abs \longleftrightarrow ?conc$ )

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-merge-abs-cp-right-compatible}$ :

$\text{cdcl}_W\text{-merge-abs-cp } S \ V \implies \text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv (state } S) \implies$

$V \sim W \implies \text{cdcl}_W\text{-merge-abs-cp } S \ W$

$\langle \text{proof} \rangle$

**interpretation**  $\text{cdcl}_W\text{-merge-abs-cp}$ :  $\text{relation-implied-relation-abs}$

$\text{cdcl}_W\text{-mset.cdcl}_W\text{-merge-cp } \text{cdcl}_W\text{-merge-abs-cp state } \text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv}$

$\langle \text{proof} \rangle$

**inductive**  $\text{cdcl}_W\text{-merge-abs-stgy}$  **for**  $S :: 'st$  **where**

$\text{fw-s-cp}$ :  $\text{full1 } \text{cdcl}_W\text{-merge-abs-cp } S \ T \implies \text{cdcl}_W\text{-merge-abs-stgy } S \ T \mid$

$\text{fw-s-decide}$ :  $\text{decide-abs } S \ T \implies \text{no-step } \text{cdcl}_W\text{-merge-abs-cp } S \implies \text{full } \text{cdcl}_W\text{-merge-abs-cp } T \ U$

$\implies \text{cdcl}_W\text{-merge-abs-stgy } S \ U$

**lemma**  $\text{cdcl}_W\text{-cp-cdcl}_W\text{-abs-cp}$ :

**assumes**  $\text{stgy} : \text{cdcl}_W\text{-merge-abs-stgy } S \ T$  **and**

$\text{inv} : \text{cdcl}_W\text{-mset.cdcl}_W\text{-all-struct-inv (state } S)$

**shows**  $\text{cdcl}_W\text{-mset.cdcl}_W\text{-merge-stgy (state } S) \text{ (state } T)$

$\langle \text{proof} \rangle$

**lemma**  $\text{cdcl}_W\text{-merge-abs-stgy-exists-cdcl}_W\text{-merge-stgy}$ :

**assumes**  
*inv*:  $cdcl_W\text{-mset}.cdcl_W\text{-all-struct-inv } S$  **and**  
*SS'*:  $S \sim_m \text{state } S'$  **and**  
*st*:  $cdcl_W\text{-mset}.cdcl_W\text{-merge-stgy } S \ T$   
**shows**  $\exists U. \ cdcl_W\text{-merge-abs-stgy } S' \ U \wedge T \sim_m \text{state } U$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-abs-stgy-right-compatible*:

**assumes**  
*inv*:  $cdcl_W\text{-mset}.cdcl_W\text{-all-struct-inv } (\text{state } S)$  **and**  
*st*:  $cdcl_W\text{-merge-abs-stgy } S \ T$  **and**  
*TU*:  $T \sim V$   
**shows**  $cdcl_W\text{-merge-abs-stgy } S \ V$   
 $\langle \text{proof} \rangle$

**interpretation** *cdcl<sub>W</sub>-merge-abs-stgy: relation-implied-relation-abs*

$cdcl_W\text{-mset}.cdcl_W\text{-merge-stgy } cdcl_W\text{-merge-abs-stgy } \text{state } cdcl_W\text{-mset}.cdcl_W\text{-all-struct-inv}$   
 $\langle \text{proof} \rangle$

**lemma** *cdcl<sub>W</sub>-merge-abs-stgy-final-State-conclusive*:

**fixes**  $T :: 'st$   
**assumes**  
*full*:  $full \ cdcl_W\text{-merge-abs-stgy } (\text{conc-init-state } N) \ T$  **and**  
*n-d*:  $distinct\text{-mset-mset } (mset\text{-class } N)$   
**shows**  $(\text{conc-conflicting } T = \text{Some } \{\#\} \wedge \text{unsatisfiable } (set\text{-mset } (mset\text{-class } N)))$   
 $\vee (\text{conc-conflicting } T = \text{None} \wedge \text{conc-trail } T \models_{asm} mset\text{-class } N$   
 $\wedge \text{satisfiable } (set\text{-mset } (mset\text{-class } N)))$   
 $\langle \text{proof} \rangle$

**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 is the principle behind the two-watched literals, an implementation has 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 algorithm.

**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*  $\Rightarrow$  (*'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*  $\equiv$  *map mmset-of-mlit (raw-trail S)*

**abbreviation** *clauses-of-l* **where**

*clauses-of-l*  $\equiv$   $\lambda L. \text{mset } (\text{map } \text{mset } L)$

**definition** *raw-clause* :: *'v twl-clause*  $\Rightarrow$  *'v literal list* **where**

*raw-clause C*  $\equiv$  *watched C @ unwatched C*

**definition** *clause* :: *'v twl-clause*  $\Rightarrow$  *'v clause* **where**

*clause C*  $\equiv$  *mset (raw-clause C)*

**lemma** *clause-def-lambda*:

*clause* = ( $\lambda C. \text{mset } (\text{raw-clause } C)$ )

*<proof>*

**abbreviation** *raw-clss* :: *'v twl-state*  $\Rightarrow$  *'v clauses* **where**

*raw-clss S*  $\equiv$  *mset (map clause (raw-init-clss S @ raw-learned-clss S))*

**abbreviation** *raw-clss-l* :: *'a twl-clause list*  $\Rightarrow$  *'a literal multiset multiset* **where**

*raw-clss-l C*  $\equiv$  *mset (map clause C)*

**interpretation** *raw-cl* *clause* *<proof>*

**lemma** *mset-map-clause-remove1-cond*:

*mset (map ( $\lambda x. \text{mset } (\text{unwatched } x) + \text{mset } (\text{watched } x)$ )*  
*(remove1-cond ( $\lambda D. \text{clause } D = \text{clause } a$ ) Cs)) =*  
*remove1-mset (clause a) (mset (map clause Cs))*  
*<proof>*

**interpretation** *raw-clss*

*clause*

*raw-clss-l op @*

$\lambda L C. L \in \text{set } C \text{ op } \# \lambda C. \text{remove1-cond } (\lambda D. \text{clause } D = \text{clause } C)$

$\langle \text{proof} \rangle$

**lemma** *ex-mset-unwatched-watched:*

$\exists a. \text{mset } (\text{unwatched } a) + \text{mset } (\text{watched } a) = E$

$\langle \text{proof} \rangle$

**interpretation** *twl: abs-state<sub>W</sub>-ops*

*clause*

*raw-clss-l op @*

$\lambda L C. L \in \text{set } C \text{ op } \# \lambda C. \text{remove1-cond } (\lambda D. \text{clause } D = \text{clause } C)$

$\text{mset } \lambda xs \text{ ys. case-prod append (fold } (\lambda x \text{ (ys, zs)}. (\text{remove1 } x \text{ ys, } x \# \text{zs})) \text{ xs (ys, []))$

*remove1*

*raw-clause*  $\lambda C. \text{TWL-Clause } [] C$

*trail*  $\lambda S. \text{hd } (\text{raw-trail } S)$

*raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting*

**rewrites**

*twl.mmset-of-mlit = mmset-of-mlit*

$\langle \text{proof} \rangle$

**declare** *CDCL-Two-Watched-Literals.twl.mset-ccls-ccls-of-cl[simp del]*

**definition**

*candidates-propagate*  $:: 'v \text{ twl-state} \Rightarrow ('v \text{ literal} \times 'v \text{ twl-clause}) \text{ set}$

**where**

*candidates-propagate*  $S =$

$\{(L, C) \mid L C.$

$C \in \text{set } (\text{twl.raw-clauses } S) \wedge$

$\text{set } (\text{watched } C) - (\text{uminus ' lits-of-l } (\text{trail } S)) = \{L\} \wedge$

$\text{undefined-lit } (\text{raw-trail } S) L\}$

**definition** *candidates-conflict*  $:: 'v \text{ twl-state} \Rightarrow 'v \text{ twl-clause set}$  **where**

*candidates-conflict*  $S =$

$\{C. C \in \text{set } (\text{twl.raw-clauses } S) \wedge$

$\text{set } (\text{watched } C) \subseteq \text{uminus ' lits-of-l } (\text{raw-trail } S)\}$

**primrec** (*nonexhaustive*) *index*  $:: 'a \text{ list} \Rightarrow 'a \Rightarrow \text{nat}$  **where**

*index*  $(a \# l) c = (\text{if } a = c \text{ then } 0 \text{ else } 1 + \text{index } l c)$

**lemma** *index-nth:*

$a \in \text{set } l \implies l ! (\text{index } l a) = a$

$\langle \text{proof} \rangle$

## 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-tw-l-cl*  $:: 'v \text{ twl-clause} \Rightarrow \text{bool}$  **where**

*struct-wf-tw-l-cl*  $(\text{TWL-Clause } W UW) \longleftrightarrow$

$$\text{distinct } W \wedge \text{length } W \leq 2 \wedge (\text{length } W < 2 \longrightarrow \text{set } UW \subseteq \text{set } W)$$

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  $\Rightarrow$   
           'v twl-clause  $\Rightarrow$  bool

**where**

*watched-only-lazy-updates*  $M$  (TWL-Clause  $W$   $UW$ )  $\longleftrightarrow$   
 ( $\forall L' \in \text{set } W. \forall L \in \text{set } UW.$   
 $-L' \in \text{lits-of-l } M \longrightarrow -L \in \text{lits-of-l } M \longrightarrow L \notin \text{set } W \longrightarrow$   
 $\text{index } (\text{map lit-of } M) (-L') \leq \text{index } (\text{map lit-of } M) (-L)$ )

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-cl* :: ('a, 'b) ann-lits  $\Rightarrow$  'a twl-clause  $\Rightarrow$   
 bool **where**  
*watched-wf-twl-cl*  $M$  (TWL-Clause  $W$   $UW$ )  $\longleftrightarrow$   
 ( $\forall L \in \text{set } W. -L \in \text{lits-of-l } M \longrightarrow (\forall L' \in \text{set } UW. L' \notin \text{set } W \longrightarrow -L' \in \text{lits-of-l } M)$ )

Here are the invariant strictly related to the 2-WL data structure.

**primrec** *wf-twl-cl* :: ('v, 'mark) ann-lits  $\Rightarrow$  'v twl-clause  $\Rightarrow$  bool **where**  
*wf-twl-cl*  $M$  (TWL-Clause  $W$   $UW$ )  $\longleftrightarrow$   
 $\text{struct-wf-twl-cl} (\text{TWL-Clause } W \text{ } UW) \wedge \text{watched-wf-twl-cl } M (\text{TWL-Clause } W \text{ } UW) \wedge$   
 $\text{watched-only-lazy-updates } M (\text{TWL-Clause } W \text{ } UW)$

**lemma** *wf-twl-cl-annotation-independant*:

**assumes**  $M$ :  $\text{map lit-of } M = \text{map lit-of } M'$

**shows**  $\text{wf-twl-cl } M (\text{TWL-Clause } W \text{ } UW) \longleftrightarrow \text{wf-twl-cl } M' (\text{TWL-Clause } W \text{ } UW)$

*<proof>*

**lemma** *wf-twl-cl-wf-twl-cl-tl*:

**assumes**  $\text{wf}$ :  $\text{wf-twl-cl } M \text{ } C$  **and**  $n\text{-d}$ :  $\text{no-dup } M$

**shows**  $\text{wf-twl-cl } (\text{tl } M) \text{ } C$

*<proof>*

**lemma** *wf-twl-cl-append*:

**assumes**

$n\text{-d}$ :  $\text{no-dup } (M' @ M)$  **and**

$\text{wf}$ :  $\text{wf-twl-cl } (M' @ M) \text{ } C$

**shows**  $\text{wf-twl-cl } M \text{ } C$

*<proof>*

**definition** *wf-twl-state* :: 'v twl-state  $\Rightarrow$  bool **where**

$\text{wf-twl-state } S \longleftrightarrow$

$(\forall C \in \text{set } (\text{twl.raw-clauses } S). \text{wf-twl-cl } (\text{raw-trail } S) \text{ } C) \wedge \text{no-dup } (\text{raw-trail } S)$

**lemma** *wf-candidates-propagate-sound*:

**assumes**  $\text{wf}$ :  $\text{wf-twl-state } S$  **and**

$\text{cand}$ :  $(L, C) \in \text{candidates-propagate } S$

**shows**  $\text{raw-trail } S \models_{\text{as}} C \text{Not } (\text{mset } (\text{removeAll } L (\text{raw-clause } C))) \wedge \text{undefined-lit } (\text{raw-trail } S) \text{ } L$   
 (**is** ?Not  $\wedge$  ?undef)

$\langle \text{proof} \rangle$

**lemma** *wf-candidates-propagate-complete*:

**assumes** *wf*: *wf-twl-state* *S* **and**

*c-mem*:  $C \in \text{set } (\text{twl.raw-clauses } S)$  **and**

*l-mem*:  $L \in \text{set } (\text{raw-clause } C)$  **and**

*unsat*:  $\text{trail } S \models_{\text{as}} \text{CNot } (\text{mset-set } (\text{set } (\text{raw-clause } C) - \{L\}))$  **and**

*undef*: *undefined-lit* (*raw-trail* *S*) *L*

**shows**  $(L, C) \in \text{candidates-propagate } S$

$\langle \text{proof} \rangle$

**lemma** *wf-candidates-conflict-sound*:

**assumes** *wf*: *wf-twl-state* *S* **and**

*cand*:  $C \in \text{candidates-conflict } S$

**shows**  $\text{trail } S \models_{\text{as}} \text{CNot } (\text{clause } C) \wedge C \in \text{set } (\text{twl.raw-clauses } S)$

$\langle \text{proof} \rangle$

**lemma** *wf-candidates-conflict-complete*:

**assumes** *wf*: *wf-twl-state* *S* **and**

*c-mem*:  $C \in \text{set } (\text{twl.raw-clauses } S)$  **and**

*unsat*:  $\text{trail } S \models_{\text{as}} \text{CNot } (\text{clause } C)$

**shows**  $C \in \text{candidates-conflict } S$

$\langle \text{proof} \rangle$

**typedef**  $'v \text{ wf-twl} = \{S :: 'v \text{ twl-state. wf-twl-state } S\}$

**morphisms** *rough-state-of-twl twl-of-rough-state*

$\langle \text{proof} \rangle$

**lemma** [*code abstype*]:

*twl-of-rough-state* (*rough-state-of-twl* *S*) = *S*

$\langle \text{proof} \rangle$

**lemma** *wf-twl-state-rough-state-of-twl[simp]*: *wf-twl-state* (*rough-state-of-twl* *S*)

$\langle \text{proof} \rangle$

**abbreviation** *candidates-conflict-twl* ::  $'v \text{ wf-twl} \Rightarrow 'v \text{ twl-clause set}$  **where**

*candidates-conflict-twl* *S*  $\equiv \text{candidates-conflict } (\text{rough-state-of-twl } S)$

**abbreviation** *candidates-propagate-twl* ::  $'v \text{ wf-twl} \Rightarrow ('v \text{ literal} \times 'v \text{ twl-clause}) \text{ set}$  **where**

*candidates-propagate-twl* *S*  $\equiv \text{candidates-propagate } (\text{rough-state-of-twl } S)$

**abbreviation** *raw-trail-twl* ::  $'a \text{ wf-twl} \Rightarrow ('a, 'a \text{ twl-clause}) \text{ ann-lits}$  **where**

*raw-trail-twl* *S*  $\equiv \text{raw-trail } (\text{rough-state-of-twl } S)$

**abbreviation** *trail-twl* ::  $'a \text{ wf-twl} \Rightarrow ('a, 'a \text{ literal multiset}) \text{ ann-lits}$  **where**

*trail-twl* *S*  $\equiv \text{trail } (\text{rough-state-of-twl } S)$

**abbreviation** *raw-clauses-twl* ::  $'a \text{ wf-twl} \Rightarrow 'a \text{ twl-clause list}$  **where**

*raw-clauses-twl* *S*  $\equiv \text{twl.raw-clauses } (\text{rough-state-of-twl } S)$

**abbreviation** *raw-init-clss-twl* ::  $'a \text{ wf-twl} \Rightarrow 'a \text{ twl-clause list}$  **where**

*raw-init-clss-twl* *S*  $\equiv \text{raw-init-clss } (\text{rough-state-of-twl } S)$

**abbreviation** *raw-learned-clss-twl* ::  $'a \text{ wf-twl} \Rightarrow 'a \text{ twl-clause list}$  **where**

*raw-learned-clss-twl* *S*  $\equiv \text{raw-learned-clss } (\text{rough-state-of-twl } S)$

**abbreviation** *backtrack-lvl-twl* **where**  
*backtrack-lvl-twl*  $S \equiv \text{backtrack-lvl } (\text{rough-state-of-twl } S)$

**abbreviation** *raw-conflicting-twl* **where**  
*raw-conflicting-twl*  $S \equiv \text{raw-conflicting } (\text{rough-state-of-twl } S)$

**lemma** *wf-candidates-twl-conflict-complete*:

**assumes**

*c-mem*:  $C \in \text{set } (\text{raw-clauses-twl } S)$  **and**

*unsat*:  $\text{trail-twl } S \models_{\text{as}} \text{CNot } (\text{clause } C)$

**shows**  $C \in \text{candidates-conflict-twl } S$

*<proof>*

**abbreviation** *update-backtrack-lvl* **where**

*update-backtrack-lvl*  $k \ S \equiv$

$\text{TWL-State } (\text{raw-trail } S) (\text{raw-init-clss } S) (\text{raw-learned-clss } S) \ k \ (\text{raw-conflicting } S)$

**abbreviation** *update-conflicting* **where**

*update-conflicting*  $C \ S \equiv$

$\text{TWL-State } (\text{raw-trail } S) (\text{raw-init-clss } S) (\text{raw-learned-clss } S) (\text{backtrack-lvl } S) \ C$

## Abstract 2-WL

**definition** *tl-trail* **where**

*tl-trail*  $S =$

$\text{TWL-State } (\text{tl } (\text{raw-trail } S)) (\text{raw-init-clss } S) (\text{raw-learned-clss } S) (\text{backtrack-lvl } S)$   
 $(\text{raw-conflicting } S)$

**locale** *abstract-twl* =

**fixes**

*watch* ::  $'v \ \text{twl-state} \Rightarrow 'v \ \text{literal list} \Rightarrow 'v \ \text{twl-clause}$  **and**

*rewatch* ::  $'v \ \text{literal} \Rightarrow 'v \ \text{twl-state} \Rightarrow$

$'v \ \text{twl-clause} \Rightarrow 'v \ \text{twl-clause}$  **and**

*restart-learned* ::  $'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-clause list}$

**assumes**

*clause-watch*:  $\text{no-dup } (\text{raw-trail } S) \Longrightarrow \text{clause } (\text{watch } S \ C) = \text{mset } C$  **and**

*wf-watch*:  $\text{no-dup } (\text{raw-trail } S) \Longrightarrow \text{wf-twl-cls } (\text{raw-trail } S) (\text{watch } S \ C)$  **and**

*clause-rewatch*:  $\text{clause } (\text{rewatch } L' \ S \ C') = \text{clause } C'$  **and**

*wf-rewatch*:

$\text{no-dup } (\text{raw-trail } S) \Longrightarrow \text{undefined-lit } (\text{raw-trail } S) (\text{lit-of } L) \Longrightarrow$

$\text{wf-twl-cls } (\text{raw-trail } S) \ C' \Longrightarrow$

$\text{wf-twl-cls } (L \ \# \ \text{raw-trail } S) (\text{rewatch } (\text{lit-of } L) \ S \ C')$

**and**

*restart-learned*:  $\text{mset } (\text{restart-learned } S) \subseteq \# \ \text{mset } (\text{raw-learned-clss } S)$  — We need *mset* and not *set* to take care of duplicates.

**begin**

**definition**

*cons-trail* ::  $( 'v, 'v \ \text{twl-clause} ) \ \text{ann-lit} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

**where**

*cons-trail*  $L \ S =$

$\text{TWL-State } (L \ \# \ \text{raw-trail } S) (\text{map } (\text{rewatch } (\text{lit-of } L) \ S) (\text{raw-init-clss } S))$

$(\text{map } (\text{rewatch } (\text{lit-of } L) \ S) (\text{raw-learned-clss } S)) (\text{backtrack-lvl } S) (\text{raw-conflicting } S)$

**definition**

*add-init-cls* ::  $'v \ \text{literal list} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

**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**

$add-learned-cls :: 'v \ literal \ list \Rightarrow 'v \ twl-state \Rightarrow 'v \ twl-state$

**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**

$remove-cls :: 'v \ literal \ list \Rightarrow 'v \ twl-state \Rightarrow 'v \ twl-state$

**where**

$remove-cls \ C \ S =$   
 $TWL-State \ (raw-trail \ S)$   
 $(removeAll-cond \ (\lambda D. \ clause \ D = mset \ C) \ (raw-init-clss \ S))$   
 $(removeAll-cond \ (\lambda D. \ clause \ D = mset \ C) \ (raw-learned-clss \ S))$   
 $(backtrack-lvl \ S)$   
 $(raw-conflicting \ S)$

**definition**  $init-state :: 'v \ literal \ list \Rightarrow 'v \ twl-state$  **where**

$init-state \ N = fold \ add-init-cls \ N \ (TWL-State \ [] \ [] \ 0 \ None)$

**lemma** *unchanged-fold-add-init-cls:*

$raw-trail \ (fold \ add-init-cls \ Cs \ (TWL-State \ M \ N \ U \ k \ C)) = M$   
 $raw-learned-clss \ (fold \ add-init-cls \ Cs \ (TWL-State \ M \ N \ U \ k \ C)) = U$   
 $backtrack-lvl \ (fold \ add-init-cls \ Cs \ (TWL-State \ M \ N \ U \ k \ C)) = k$   
 $raw-conflicting \ (fold \ add-init-cls \ Cs \ (TWL-State \ M \ N \ U \ k \ C)) = C$   
 $\langle proof \rangle$

**lemma** *unchanged-init-state[simp]:*

$raw-trail \ (init-state \ N) = []$   
 $raw-learned-clss \ (init-state \ N) = []$   
 $backtrack-lvl \ (init-state \ N) = 0$   
 $raw-conflicting \ (init-state \ N) = None$   
 $\langle proof \rangle$

**lemma** *clauses-init-fold-add-init:*

$no-dup \ M \Longrightarrow$   
 $twl.conc-init-clss \ (fold \ add-init-cls \ Cs \ (TWL-State \ M \ N \ U \ k \ C)) =$   
 $clauses-of-l \ Cs + raw-clss-l \ N$   
 $\langle proof \rangle$

**lemma** *init-clss-init-state[simp]:*  $twl.conc-init-clss \ (init-state \ N) = clauses-of-l \ N$

$\langle proof \rangle$

**definition**  $restart'$  **where**

$restart' \ S = TWL-State \ [] \ (raw-init-clss \ S) \ (restart-learned \ S) \ 0 \ None$

**end**

## Instanciation of the previous locale

**definition**  $watch-nat :: 'v \ twl-state \Rightarrow 'v \ literal \ list \Rightarrow 'v \ twl-clause$  **where**



```

watch-nat S C =
  (let
    C' = remdups C;
    neg-not-assigned = filter (λL. -L ∉ lits-of-l (raw-trail S)) C';
    neg-assigned-sorted-by-trail = filter (λL. L ∈ set C) (map (λL. -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**  $l :: 'a \text{ list}$   
**assumes**  
 $l = [] \implies P$  **and**  
 $\bigwedge x. l = [x] \implies P$  **and**  
 $\bigwedge x \ y \ xs. l = x \# y \# xs \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *filter-in-list-prop-verifiedD*:  
**assumes**  $[L \leftarrow P \ . \ Q \ L] = l$   
**shows**  $\forall x \in \text{set } l. x \in \text{set } P \wedge Q \ x$   
 $\langle \text{proof} \rangle$

**lemma** *no-dup-filter-diff*:  
**assumes**  $n\text{-d: no-dup } M$  **and**  $H: [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) \ M. L \in \text{set } C] = l$   
**shows** *distinct*  $l$   
 $\langle \text{proof} \rangle$

**lemma** *watch-nat-lists-disjointD*:  
**assumes**  
 $l: [L \leftarrow \text{remdups } C. - L \notin \text{lits-of-l } (\text{raw-trail } S)] = l$  **and**  
 $l': [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{raw-trail } S) \ . \ L \in \text{set } C] = l'$   
**shows**  $\forall x \in \text{set } l. \forall y \in \text{set } l'. x \neq y$   
 $\langle \text{proof} \rangle$

**lemma** *watch-nat-list-cases-witness*[*consumes 2, case-names Nil-Nil Nil-single Nil-other single-Nil single-other other*]:  
**fixes**  
 $C :: 'v \text{ literal list}$  **and**  
 $S :: 'v \text{ twl-state}$   
**defines**  
 $xs \equiv [L \leftarrow \text{remdups } C. - L \notin \text{lits-of-l } (\text{raw-trail } S)]$  **and**  
 $ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{raw-trail } S) \ . \ L \in \text{set } C]$   
**assumes**  
 $n\text{-d: no-dup } (\text{raw-trail } S)$  **and**  
 $\text{Nil-Nil: } xs = [] \implies ys = [] \implies P$  **and**  
 $\text{Nil-single:}$   
 $\bigwedge a. xs = [] \implies ys = [a] \implies a \in \text{set } C \implies P$  **and**  
 $\text{Nil-other: } \bigwedge a \ b \ ys'. xs = [] \implies ys = a \# b \# ys' \implies a \neq b \implies P$  **and**  
 $\text{single-Nil: } \bigwedge a. xs = [a] \implies ys = [] \implies P$  **and**  
 $\text{single-other: } \bigwedge a \ b \ ys'. xs = [a] \implies ys = b \# ys' \implies a \neq b \implies P$  **and**  
 $\text{other: } \bigwedge a \ b \ xs'. xs = a \# b \# xs' \implies a \neq b \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *watch-nat-list-cases* [*consumes 1, case-names Nil-Nil Nil-single Nil-other single-Nil*]

*single-other other*]:

**fixes**

$C :: 'v \text{ literal list}$  **and**

$S :: 'v \text{ twl-state}$

**defines**

$xs \equiv [L \leftarrow \text{remdups } C . - L \notin \text{lits-of-l (raw-trail } S)]$  **and**

$ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{raw-trail } S) . L \in \text{set } C]$

**assumes**

*n-d*: *no-dup* (raw-trail *S*) **and**

*Nil-Nil*:  $xs = [] \implies ys = [] \implies P$  **and**

*Nil-single*:

$\bigwedge a. xs = [] \implies ys = [a] \implies a \in \text{set } C \implies P$  **and**

*Nil-other*:  $\bigwedge a \ b \ ys'. xs = [] \implies ys = a \# b \# ys' \implies a \neq b \implies P$  **and**

*single-Nil*:  $\bigwedge a. xs = [a] \implies ys = [] \implies P$  **and**

*single-other*:  $\bigwedge a \ b \ ys'. xs = [a] \implies ys = b \# ys' \implies a \neq b \implies P$  **and**

*other*:  $\bigwedge a \ b \ xs'. xs = a \# b \# xs' \implies a \neq b \implies P$

**shows** *P*

*<proof>*

**lemma** *watch-nat-lists-set-union-witness*:

**fixes**

$C :: 'v \text{ literal list}$  **and**

$S :: 'v \text{ twl-state}$

**defines**

$xs \equiv [L \leftarrow \text{remdups } C . - L \notin \text{lits-of-l (raw-trail } S)]$  **and**

$ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{raw-trail } S) . L \in \text{set } C]$

**assumes** *n-d*: *no-dup* (raw-trail *S*)

**shows**  $\text{set } C = \text{set } xs \cup \text{set } ys$

*<proof>*

**lemma** *mset-intersection-inclusion*:  $A + (B - A) = B \longleftrightarrow A \subseteq \# B$

*<proof>*

**lemma** *clause-watch-nat*:

**assumes** *no-dup* (raw-trail *S*)

**shows**  $\text{clause (watch-nat } S \ C) = \text{mset } C$

*<proof>*

**lemma** *index-uminus-index-map-uminus*:

$-a \in \text{set } L \implies \text{index } L \ (-a) = \text{index (map uminus } L) \ (a::'a \text{ literal})$

*<proof>*

**lemma** *index-filter*:

$a \in \text{set } L \implies b \in \text{set } L \implies P \ a \implies P \ b \implies$

$\text{index } L \ a \leq \text{index } L \ b \longleftrightarrow \text{index (filter } P \ L) \ a \leq \text{index (filter } P \ L) \ b$

*<proof>*

**lemma** *foldr-remove1-W-Nil[simp]*:  $\text{foldr remove1 } W \ [] = []$

*<proof>*

**lemma** *image-lit-of-mmset-of-mlit[simp]*:

$\text{lit-of } ' \text{ mmset-of-mlit } ' \ A = \text{lit-of } ' \ A$

*<proof>*

**lemma** *distinct-filter-eq*:

**assumes** *distinct* *xs*

**shows**  $[L \leftarrow xs. L = a] = (\text{if } a \in \text{set } xs \text{ then } [a] \text{ else } [])$   
 $\langle \text{proof} \rangle$

**lemma** *no-dup-distinct-map-uminus-lit-of*:  
 $\text{no-dup } xs \implies \text{distinct } (\text{map } (\lambda L. - \text{lit-of } L) \text{ } xs)$   
 $\langle \text{proof} \rangle$

**lemma** *wf-watch-witness*:  
**fixes**  $C :: 'v \text{ literal list}$  **and**  
 $S :: 'v \text{ twl-state}$   
**defines**  
 $\text{ass: neg-not-assigned} \equiv \text{filter } (\lambda L. -L \notin \text{lits-of-l } (\text{raw-trail } S)) (\text{remdups } C)$  **and**  
 $\text{tr: neg-assigned-sorted-by-trail} \equiv \text{filter } (\lambda L. L \in \text{set } C) (\text{map } (\lambda L. -\text{lit-of } L) (\text{raw-trail } S))$   
**defines**  
 $W: W \equiv \text{take } 2 (\text{neg-not-assigned } @ \text{neg-assigned-sorted-by-trail})$   
**assumes**  
 $n\text{-d}[\text{simp}]: \text{no-dup } (\text{raw-trail } S)$   
**shows**  $\text{wf-twlc} (\text{raw-trail } S) (\text{TWL-Clause } W (\text{foldr remove1 } W C))$   
 $\langle \text{proof} \rangle$

**lemma** *wf-watch-nat*:  $\text{no-dup } (\text{raw-trail } S) \implies \text{wf-twlc} (\text{raw-trail } S) (\text{watch-nat } S C)$   
 $\langle \text{proof} \rangle$

**definition**  
 $\text{rewatch-nat} ::$   
 $'v \text{ literal} \Rightarrow 'v \text{ twl-state} \Rightarrow 'v \text{ twl-clause} \Rightarrow 'v \text{ twl-clause}$   
**where**  
 $\text{rewatch-nat } L S C =$   
 $(\text{if } -L \in \text{set } (\text{watched } C) \text{ then}$   
 $\text{case filter } (\lambda L'. L' \notin \text{set } (\text{watched } C) \wedge -L' \notin \text{insert } L (\text{lits-of-l } (\text{trail } S)))$   
 $(\text{unwatched } C) \text{ of}$   
 $[] \Rightarrow C$   
 $| L' \# - \Rightarrow$   
 $\text{TWL-Clause } (L' \# \text{remove1 } (-L) (\text{watched } C)) (-L \# \text{remove1 } L' (\text{unwatched } C))$   
 $\text{else}$   
 $C)$

**lemma** *clause-rewatch-nat*:  
**fixes**  $UW :: 'v \text{ literal list}$  **and**  
 $S :: 'v \text{ twl-state}$  **and**  
 $L :: 'v \text{ literal}$  **and**  $C :: 'v \text{ twl-clause}$   
**shows**  $\text{clause } (\text{rewatch-nat } L S C) = \text{clause } C$   
 $\langle \text{proof} \rangle$

**lemma** *filter-sorted-list-of-multiset-Nil*:  
 $[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = [] \longleftrightarrow (\forall x \in \# M. \neg p \ x)$   
 $\langle \text{proof} \rangle$

**lemma** *filter-sorted-list-of-multiset-ConsD*:  
 $[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = x \# xs \implies p \ x$   
 $\langle \text{proof} \rangle$

**lemma** *mset-minus-single-eq-empty*:  
 $a - \{\#b\} = \{\#\} \longleftrightarrow a = \{\#b\} \vee a = \{\#\}$   
 $\langle \text{proof} \rangle$

```

lemma size-mset-le-2-cases:
  assumes size  $W \leq 2$ 
  shows  $W = \{\#\} \vee (\exists a. W = \{\#a\# \}) \vee (\exists a\ b. W = \{\#a,b\# \})$ 
  <proof>

lemma filter-sorted-list-of-multiset-eqD:
  assumes  $[x \leftarrow \text{sorted-list-of-multiset } A. p\ x] = x \# xs$  (is ?comp = -)
  shows  $x \in\# A$ 
  <proof>

lemma clause-rewatch-witness':
  assumes
    wf: wf-tw-l-cls (raw-trail S) C and
    undef: undefined-lit (raw-trail S) (lit-of L)
  shows wf-tw-l-cls (L # raw-trail S) (rewatch-nat (lit-of L) S C)
  <proof>

interpretation twl: abstract-tw-l watch-nat rewatch-nat raw-learned-clss
  <proof>

interpretation twl2: abstract-tw-l watch-nat rewatch-nat  $\lambda\cdot. []$ 
  <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<sub>W</sub>.cdcl<sub>W</sub>*

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 data-structure.

```

context abstract-tw-l
begin

```

**Direct Interpretation** **lemma** *mset-map-removeAll-cond*:

```

  mset (map clause
    (removeAll-cond ( $\lambda D. \text{clause } D = \text{clause } C$ ) N))
  = mset (removeAll (clause C) (map clause N))
  <proof>

```

**lemma** *mset-raw-init-clss-init-state*:

```

  mset (map clause (raw-init-clss (init-state (map raw-clause N))))
  = mset (map clause N)
  <proof>

```

```

fun reduce-trail-to where
  reduce-trail-to M1 S =
    (case S of

```

$(TWL\text{-}State\ M\ N\ U\ k\ C) \Rightarrow TWL\text{-}State\ (drop\ (length\ M - length\ M1)\ M)\ N\ U\ k\ C)$

**interpretation** *rough-cdcl*: *abs-state<sub>W</sub>-ops*

*clause*

*raw-clss-l op @*

$\lambda L\ C. L \in set\ C\ op\ \# \lambda C. remove1\text{-}cond\ (\lambda D. clause\ D = clause\ C)$

*mset*  $\lambda xs\ ys. case\text{-}prod\ append\ (fold\ (\lambda x\ (ys, zs). (remove1\ x\ ys, x\ \# zs))\ xs\ (ys, []))$   
*remove1*

*raw-clause*  $\lambda C. TWL\text{-}Clause\ []\ C$

*trail*  $\lambda S. hd\ (raw\text{-}trail\ S)$

*raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting*

*cons-trail tl-trail*  $\lambda C. add\text{-}learned\text{-}cls\ (raw\text{-}clause\ C)$

$\lambda C. remove\text{-}cls\ (raw\text{-}clause\ C)$

*update-backtrack-lvl*

*update-conflicting reduce-trail-to*  $\lambda N. init\text{-}state\ (map\ raw\text{-}clause\ N)\ restart'$

**rewrites**

*rough-cdcl.mset-of-mlit* = *mmset-of-mlit*

*<proof>*

**interpretation** *rough-cdcl*: *abs-state<sub>W</sub>*

*clause*

*raw-clss-l op @*

$\lambda L\ C. L \in set\ C\ op\ \# \lambda C. remove1\text{-}cond\ (\lambda D. clause\ D = clause\ C)$

*mset*  $\lambda xs\ ys. case\text{-}prod\ append\ (fold\ (\lambda x\ (ys, zs). (remove1\ x\ ys, x\ \# zs))\ xs\ (ys, []))$   
*remove1*

*raw-clause*  $\lambda C. TWL\text{-}Clause\ []\ C$

*trail*  $\lambda S. hd\ (raw\text{-}trail\ S)$

*raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting*

*cons-trail tl-trail*  $\lambda C. add\text{-}learned\text{-}cls\ (raw\text{-}clause\ C)$

$\lambda C. remove\text{-}cls\ (raw\text{-}clause\ C)$

*update-backtrack-lvl*

*update-conflicting reduce-trail-to*  $\lambda N. init\text{-}state\ (map\ raw\text{-}clause\ N)\ restart'$

*<proof>*

**interpretation** *rough-cdcl*: *abs-conflict-driven-clause-learning<sub>W</sub>*

*clause*

*raw-clss-l op @*

$\lambda L\ C. L \in set\ C\ op\ \# \lambda C. remove1\text{-}cond\ (\lambda D. clause\ D = clause\ C)$

*mset*  $\lambda xs\ ys. case\text{-}prod\ append\ (fold\ (\lambda x\ (ys, zs). (remove1\ x\ ys, x\ \# zs))\ xs\ (ys, []))$   
*remove1*

$\lambda C. raw\text{-}clause\ C\ \lambda C. TWL\text{-}Clause\ []\ C$

*trail*  $\lambda S. hd\ (raw\text{-}trail\ S)$

*raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting*

*cons-trail tl-trail*  $\lambda C. add\text{-}learned\text{-}cls\ (raw\text{-}clause\ C)$

$\lambda C. remove\text{-}cls\ (raw\text{-}clause\ C)$

*update-backtrack-lvl*

*update-conflicting reduce-trail-to*  $\lambda N. init\text{-}state\ (map\ raw\text{-}clause\ N)\ restart'$

*<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  $\Rightarrow$  'v wf-twl  $\Rightarrow$  'v wf-twl  
**where**  
*cons-trail-twl* L S  $\equiv$  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)

$\langle$ proof $\rangle$

**lemma** *rough-state-of-twl-cons-trail*:

undefined-lit (raw-trail-twl S) (lit-of L)  $\implies$

rough-state-of-twl (cons-trail-twl L S) = cons-trail L (rough-state-of-twl S)

$\langle$ proof $\rangle$

**abbreviation** *add-init-cls-twl* **where**

*add-init-cls-twl* C S  $\equiv$  twl-of-rough-state (add-init-cls C (rough-state-of-twl S))

**lemma** *wf-twl-add-init-cls*: wf-twl-state S  $\implies$  wf-twl-state (add-init-cls L S)

$\langle$ proof $\rangle$

**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)

$\langle$ proof $\rangle$

**abbreviation** *add-learned-cls-twl* **where**

*add-learned-cls-twl* C S  $\equiv$  twl-of-rough-state (add-learned-cls C (rough-state-of-twl S))

**lemma** *wf-twl-add-learned-cls*: wf-twl-state S  $\implies$  wf-twl-state (add-learned-cls L S)

$\langle$ proof $\rangle$

**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)

$\langle$ proof $\rangle$

**abbreviation** *remove-cls-twl* **where**

*remove-cls-twl* C S  $\equiv$  twl-of-rough-state (remove-cls C (rough-state-of-twl S))

**lemma** *set-removeAll-condD*:  $x \in \text{set } (\text{removeAll-cond } f \text{ } xs) \implies x \in \text{set } xs$

$\langle$ proof $\rangle$

**lemma** *wf-twl-remove-cls*: wf-twl-state S  $\implies$  wf-twl-state (remove-cls L S)

$\langle$ proof $\rangle$

**lemma** *rough-state-of-twl-remove-cls*:

rough-state-of-twl (remove-cls-twl L S) = remove-cls L (rough-state-of-twl S)

$\langle$ proof $\rangle$

**abbreviation** *init-state-twl* **where**

*init-state-twl* N  $\equiv$  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-cl*s *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*  $\equiv$  *twl-of-rough-state* (*tl-trail* (*rough-state-of-twl S*))

**lemma** *wf-twl-state-tl-trail*: *wf-twl-state S*  $\implies$  *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*  $\equiv$  *twl-of-rough-state* (*update-backtrack-lvl k* (*rough-state-of-twl S*))

**lemma** *wf-twl-state-update-backtrack-lvl*:  
*wf-twl-state S*  $\implies$  *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*  $\equiv$  *twl-of-rough-state* (*update-conflicting k* (*rough-state-of-twl S*))

**abbreviation** *reduce-trail-to-twl* **where**  
*reduce-trail-to-twl M1 S*  $\equiv$  *twl-of-rough-state* (*reduce-trail-to M1* (*rough-state-of-twl S*))

**lemma** *wf-twl-state-update-conflicting*:  
*wf-twl-state S*  $\implies$  *wf-twl-state* (*update-conflicting k S*)  
⟨*proof*⟩

**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*  $\equiv$  *twl.raw-clauses* (*rough-state-of-twl S*)

**abbreviation** *restart-twl* **where**  
*restart-twl S*  $\equiv$  *twl-of-rough-state* (*restart'* (*rough-state-of-twl S*))

**lemma** *mset-union-mset-setD*:

$mset\ A \subseteq\# mset\ B \implies set\ A \subseteq set\ B$   
 $\langle proof \rangle$

**lemma** *wf-wf-restart'*:  $wf\text{-}twl\text{-}state\ S \implies wf\text{-}twl\text{-}state\ (restart'\ S)$

$\langle proof \rangle$

**lemma** *rough-state-of-twl-restart-twl*:

$rough\text{-}state\text{-}of\text{-}twl\ (restart\text{-}twl\ S) = restart'\ (rough\text{-}state\text{-}of\text{-}twl\ S)$   
 $\langle proof \rangle$

**lemma** *undefined-lit-trail-twl-raw-trail*[*iff*]:

$undefined\text{-}lit\ (trail\text{-}twl\ S)\ L \longleftrightarrow undefined\text{-}lit\ (raw\text{-}trail\text{-}twl\ S)\ L$   
 $\langle proof \rangle$

**lemma** *wf-twl-reduce-trail-to*:

**assumes**  $trail\ S = M2\ @\ M1$  **and**  $wf: wf\text{-}twl\text{-}state\ S$   
**shows**  $wf\text{-}twl\text{-}state\ (reduce\text{-}trail\text{-}to\ M1\ S)$

$\langle proof \rangle$

**lemma** *trail-twl-twl-rough-state-reduce-trail-to*:

**assumes**  $trail\text{-}twl\ st = M2\ @\ M1$   
**shows**  $trail\text{-}twl\ (twl\text{-}of\text{-}rough\text{-}state\ (reduce\text{-}trail\text{-}to\ M1\ (rough\text{-}state\text{-}of\text{-}twl\ st))) = M1$   
 $\langle proof \rangle$

**lemma** *twl-of-rough-state-reduce-trail-to*:

**assumes**  $trail\text{-}twl\ st = M2\ @\ M1$  **and**

$S: rough\text{-}cdcl.state\ (rough\text{-}state\text{-}of\text{-}twl\ st) = (M, S)$

**shows**

$rough\text{-}cdcl.state$   
 $(rough\text{-}state\text{-}of\text{-}twl\ (twl\text{-}of\text{-}rough\text{-}state\ (reduce\text{-}trail\text{-}to\ M1\ (rough\text{-}state\text{-}of\text{-}twl\ st)))) =$   
 $(M1, S)\ (\text{is } ?st)\ \text{and}$   
 $raw\text{-}init\text{-}clss\text{-}twl\ (twl\text{-}of\text{-}rough\text{-}state\ (reduce\text{-}trail\text{-}to\ M1\ (rough\text{-}state\text{-}of\text{-}twl\ st)))$   
 $= raw\text{-}init\text{-}clss\text{-}twl\ st\ (\text{is } ?A)\ \text{and}$   
 $raw\text{-}learned\text{-}clss\text{-}twl\ (twl\text{-}of\text{-}rough\text{-}state\ (reduce\text{-}trail\text{-}to\ M1\ (rough\text{-}state\text{-}of\text{-}twl\ st)))$   
 $= raw\text{-}learned\text{-}clss\text{-}twl\ st\ (\text{is } ?B)\ \text{and}$   
 $backtrack\text{-}lwl\text{-}twl\ (twl\text{-}of\text{-}rough\text{-}state\ (reduce\text{-}trail\text{-}to\ M1\ (rough\text{-}state\text{-}of\text{-}twl\ st)))$   
 $= backtrack\text{-}lwl\text{-}twl\ st\ (\text{is } ?C)\ \text{and}$   
 $rough\text{-}cdcl.conc\text{-}conflicting\ (rough\text{-}state\text{-}of\text{-}twl\ (twl\text{-}of\text{-}rough\text{-}state$   
 $(reduce\text{-}trail\text{-}to\ M1\ (rough\text{-}state\text{-}of\text{-}twl\ st))))$   
 $= rough\text{-}cdcl.conc\text{-}conflicting\ (rough\text{-}state\text{-}of\text{-}twl\ st)\ (\text{is } ?D)$

$\langle proof \rangle$

**sublocale** *wf-twl: abs-conflict-driven-clause-learning<sub>w</sub>*

*clause*

$raw\text{-}clss\text{-}l\ op\ @$

$\lambda L\ C. L \in set\ C\ op\ \# \lambda C. remove1\text{-}cond\ (\lambda D. clause\ D = clause\ C)$

$mset\ \lambda xs\ ys. case\text{-}prod\ append\ (fold\ (\lambda x\ (ys, zs). (remove1\ x\ ys, x\ \# zs))\ xs\ (ys, []))$   
 $remove1$

$\lambda C. raw\text{-}clause\ C\ \lambda C. TWL\text{-}Clause\ []\ C$

$trail\text{-}twl\ \lambda S. hd\ (raw\text{-}trail\text{-}twl\ S)$

$raw\text{-}init\text{-}clss\text{-}twl$

$raw\text{-}learned\text{-}clss\text{-}twl$



*backtrack-lvl-twl*  
*raw-conflicting-twl*  
*cons-trail-twl*  
*tl-trail-twl*  
 $\lambda C. \text{add-learned-cls-twl } (\text{raw-clause } C)$   
 $\lambda C. \text{remove-cls-twl } (\text{raw-clause } C)$   
*update-backtrack-lvl-twl*  
*update-conflicting-twl*  
*reduce-trail-to-twl*  
 $\lambda N. \text{init-state-twl } (\text{map raw-clause } N)$   
*restart-twl*  
 <proof>

**declare** *local.rough-cdcl.mset-ccls-ccls-of-cls*[simp del]

**abbreviation** *state-eq-twl* (**infix**  $\sim_{TWL}$  51) **where**

*state-eq-twl*  $S \ S' \equiv \text{rough-cdcl.state-eq } (\text{rough-state-of-twl } S) (\text{rough-state-of-twl } S')$

**notation** *wf-twl.state-eq* (**infix**  $\sim$  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*

$:: 'v \text{ wf-twl} \Rightarrow 'v \text{ wf-twl} \Rightarrow \text{bool}$  **where**

*propagate-twl-rule*:  $(L, C) \in \text{candidates-propagate-twl } S \Rightarrow$

$S' \sim \text{cons-trail-twl } (\text{Propagated } L \ C) \ S \Rightarrow$

$\text{raw-conflicting-twl } S = \text{None} \Rightarrow$

$\text{propagate-twl } S \ S'$

**inductive-cases** *propagate-twlE*: *propagate-twl*  $S \ T$

**lemma** *propagate-twl-iff-propagate*:

**assumes** *inv*: *cdcl<sub>W</sub>-mset.cdcl<sub>W</sub>-all-struct-inv* (*wf-twl.state*  $S$ )

**shows** *wf-twl.propagate-abs*  $S \ T \longleftrightarrow \text{propagate-twl } S \ T$  (**is**  $?P \longleftrightarrow ?T$ )

<proof>

**no-notation** *twl.state-eq-twl* (**infix**  $\sim_{TWL}$  51)

**inductive** *conflict-twl* **where**

*conflict-twl-rule*:

$C \in \text{candidates-conflict-twl } S \Rightarrow$

$S' \sim \text{update-conflicting-twl } (\text{Some } (\text{raw-clause } C)) \ S \Rightarrow$

$\text{raw-conflicting-twl } S = \text{None} \Rightarrow$

$\text{conflict-twl } S \ S'$

**inductive-cases** *conflict-twlE*: *conflict-twl*  $S \ T$

**lemma** *conflict-twl-iff-conflict*:

**shows** *wf-twl.conflict-abs*  $S \ T \longleftrightarrow \text{conflict-twl } S \ T$  (**is**  $?C \longleftrightarrow ?T$ )

<proof>

We have shown that we we can use *conflict-twl* and *propagate-twl* in a CDCL calculus.

**end**

**end**