

# Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

February 4, 2016

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Transitions</b>   | <b>5</b>  |
| 1.1      | More theorems about Closures . . . . .                       | 5         |
| 1.2      | Full Transitions . . . . .                                   | 6         |
| 1.3      | Well-Foundedness and Full Transitions . . . . .              | 7         |
| 1.4      | More Well-Foundedness . . . . .                              | 8         |
| <b>2</b> | <b>Various Lemmas</b>  | <b>10</b> |
| <b>3</b> | <b>More List</b>   | <b>12</b> |
| 3.1      | <i>upt</i> . . . . .   | 12        |
| 3.2      | Lexicographic ordering . . . . .                             | 14        |
| <b>4</b> | <b>Logics</b>  | <b>14</b> |
| 4.1      | Definition and abstraction . . . . .                         | 14        |
| 4.2      | properties of the abstraction . . . . .                      | 16        |
| 4.3      | Subformulas and properties . . . . .                         | 18        |
| 4.4      | Positions . . . . .  | 21        |
| <b>5</b> | <b>Semantics over the syntax</b>                             | <b>24</b> |
| <b>6</b> | <b>Rewrite systems and properties</b>                        | <b>26</b> |
| 6.1      | Lifting of rewrite rules . . . . .                           | 26        |
| 6.2      | Consistency preservation . . . . .                           | 28        |
| 6.3      | Full Lifting . . . . .                                       | 29        |
| <b>7</b> | <b>Transformation testing</b>                                | <b>29</b> |
| 7.1      | Definition and first properties . . . . .                    | 29        |
| 7.2      | Invariant conservation . . . . .                             | 32        |
| 7.2.1    | Invariant while lifting of the rewriting relation . . . . .  | 33        |
| 7.2.2    | Invariant after all rewriting . . . . .                      | 34        |
| <b>8</b> | <b>Rewrite Rules</b>   | <b>35</b> |
| 8.1      | Elimination of the equivalences . . . . .                    | 36        |
| 8.2      | Eliminate Implication . . . . .                              | 37        |
| 8.3      | Eliminate all the True and False in the formula . . . . .    | 39        |
| 8.4      | PushNeg . . . . .  | 45        |
| 8.5      | Push inside . . . . .  | 50        |
| 8.5.1    | Only one type of connective in the formula (+ not) . . . . . | 59        |

|           |  |            |
|-----------|--|------------|
| 8.5.2     | Push Conjunction   | 63         |
| 8.5.3     | Push Disjunction   | 63         |
| <b>9</b>  | <b>The full transformations</b>  | <b>64</b>  |
| 9.1       | Abstract Property characterizing that only some connective are inside the others | 64         |
| 9.1.1     | Definition   | 64         |
| 9.2       | Conjunctive Normal Form  | 66         |
| 9.2.1     | Full CNF transformation  | 67         |
| 9.3       | Disjunctive Normal Form  | 67         |
| 9.3.1     | Full DNF transform   | 68         |
| <b>10</b> | <b>More aggressive simplifications: Removing true and false at the beginning</b> | <b>68</b>  |
| 10.1      | Transformation   | 68         |
| 10.2      | More invariants  | 70         |
| 10.3      | The new CNF and DNF transformation   | 74         |
| <b>11</b> | <b>Partial Clausal Logic</b>   | <b>75</b>  |
| 11.1      | Clauses  | 75         |
| 11.2      | Partial Interpretations  | 75         |
| 11.2.1    | Consistency  | 75         |
| 11.2.2    | Atoms  | 76         |
| 11.2.3    | Totality   | 78         |
| 11.2.4    | Interpretations  | 80         |
| 11.2.5    | Satisfiability   | 82         |
| 11.2.6    | Entailment for Multisets of Clauses  | 83         |
| 11.2.7    | Tautologies  | 85         |
| 11.2.8    | Entailment for clauses and propositions  | 86         |
| 11.3      | Subsumptions   | 91         |
| 11.4      | Removing Duplicates  | 92         |
| 11.5      | Set of all Simple Clauses  | 93         |
| 11.6      | Experiment: Expressing the Entailments as Locales                                | 98         |
| 11.7      | Entailment to be extended  | 99         |
| <b>12</b> | <b>Resolution</b>  | <b>100</b> |
| 12.1      | Simplification Rules   | 100        |
| 12.2      | Unconstrained Resolution   | 102        |
| 12.2.1    | Subsumption  | 102        |
| 12.3      | Inference Rule   | 103        |
| 12.4      | Lemma about the simplified state   | 118        |
| 12.5      | Resolution and Invariants  | 120        |
| 12.5.1    | Invariants   | 121        |
| 12.5.2    | well-foundedness if the relation   | 126        |
| <b>13</b> | <b>Partial Clausal Logic</b>   | <b>142</b> |
| 13.1      | Marked Literals  | 142        |
| 13.1.1    | Definition   | 142        |
| 13.1.2    | Entailment   | 143        |
| 13.1.3    | Defined and undefined literals   | 145        |
| 13.2      | Backtracking   | 146        |
| 13.3      | Decomposition with respect to the marked literals                                | 147        |

|           |   |            |
|-----------|---|------------|
| 13.4      | Negation of Clauses                           | 154        |
| 13.5      | Other   | 157        |
| <b>14</b> | <b>NOT's CDCL</b>                             | <b>159</b> |
| 14.1      | Auxiliary Lemmas and Measure                  | 159        |
| 14.2      | Initial definitions                           | 163        |
| 14.2.1    | The state                                     | 163        |
| 14.2.2    | Definition of the operation                   | 165        |
| 14.3      | DPLL with backjumping                         | 166        |
| 14.3.1    | Definition                                    | 167        |
| 14.3.2    | Basic properties                              | 167        |
| 14.3.3    | Termination                                   | 170        |
| 14.3.4    | Normal Forms                                  | 175        |
| 14.4      | CDCL  | 182        |
| 14.4.1    | Learn and Forget                              | 182        |
| 14.4.2    | Definition of CDCL                            | 183        |
| 14.5      | CDCL with invariant                           | 187        |
| 14.6      | Termination                                   | 192        |
| 14.6.1    | Restricting learn and forget                  | 192        |
| 14.7      | CDCL with restarts                            | 203        |
| 14.7.1    | Definition                                    | 203        |
| 14.7.2    | Increasing restarts                           | 204        |
| 14.8      | Merging backjump and learning                 | 211        |
| 14.8.1    | Instantiations                                | 223        |
| <b>15</b> | <b>DPLL as an instance of NOT</b>             | <b>238</b> |
| 15.1      | DPLL with simple backtrack                    | 238        |
| 15.2      | Adding restarts                               | 243        |
| <b>16</b> | <b>DPLL</b>                                   | <b>243</b> |
| 16.1      | Rules   | 243        |
| 16.2      | Invariants                                    | 244        |
| 16.3      | Termination                                   | 252        |
| 16.4      | Final States                                  | 254        |
| 16.5      | Link with NOT's DPLL                          | 256        |
| 16.5.1    | Level of literals and clauses                 | 257        |
| 16.5.2    | Properties about the levels                   | 261        |
| <b>17</b> | <b>Weidenbach's CDCL</b>                      | <b>264</b> |
| 17.1      | The State                                     | 264        |
| 17.2      | Special Instantiation: using Triples as State | 270        |
| 17.3      | CDCL Rules                                    | 270        |
| 17.4      | Invariants                                    | 276        |
| 17.4.1    | Properties of the trail                       | 276        |
| 17.4.2    | Better-Suited Induction Principle             | 280        |
| 17.4.3    | Compatibility with $op \sim$                  | 284        |
| 17.4.4    | Conservation of some Properties               | 285        |
| 17.4.5    | Learned Clause                                | 286        |
| 17.4.6    | No alien atom in the state                    | 288        |
| 17.4.7    | No duplicates all around                      | 290        |

|           |  |            |
|-----------|--|------------|
| 17.4.8    | Conflicts and co   | 291        |
| 17.4.9    | Putting all the invariants together                                | 300        |
| 17.4.10   | No tautology is learned  | 303        |
| 17.5      | CDCL Strong Completeness   | 303        |
| 17.6      | Higher level strategy  | 305        |
| 17.6.1    | Definition   | 305        |
| 17.6.2    | Invariants   | 307        |
| 17.6.3    | Literal of highest level in conflicting clauses                    | 313        |
| 17.6.4    | Literal of highest level in marked literals                        | 316        |
| 17.6.5    | Strong completeness  | 326        |
| 17.6.6    | No conflict with only variables of level less than backtrack level | 331        |
| 17.6.7    | Final States are Conclusive  | 343        |
| 17.7      | Termination  | 349        |
| 17.8      | No Relearning of a clause  | 350        |
| 17.9      | Decrease of a measure  | 365        |
| <b>18</b> | <b>Simple Implementation of the DPLL and CDCL</b>                  | <b>371</b> |
| 18.1      | Common Rules   | 371        |
| 18.1.1    | Propagation  | 371        |
| 18.1.2    | Unit propagation for all clauses                                   | 372        |
| 18.1.3    | Decide   | 373        |
| 18.2      | Simple Implementation of DPLL                                      | 374        |
| 18.2.1    | Combining the propagate and decide: a DPLL step                    | 374        |
| 18.2.2    | Adding invariants  | 377        |
| 18.2.3    | Code export  | 383        |
| 18.3      | CDCL Implementation  | 386        |
| 18.3.1    | Definition of the rules  | 386        |
| 18.3.2    | Propagate  | 387        |
| 18.3.3    | Code generation  | 399        |
| <b>19</b> | <b>Link between Weidenbach's and NOT's CDCL</b>                    | <b>411</b> |
| 19.1      | Inclusion of the states  | 411        |
| 19.2      | More lemmas conflict-propagate and backjumping                     | 415        |
| 19.2.1    | Termination  | 415        |
| 19.2.2    | More backjumping   | 416        |
| 19.3      | CDCL FW  | 430        |
| 19.4      | FW with strategy   | 440        |
| 19.4.1    | The intermediate step  | 440        |
| 19.5      | Adding Restarts  | 476        |
| <b>20</b> | <b>Incremental SAT solving</b>                                     | <b>486</b> |
| 20.1      | We can now define the rules of the calculus                        | 525        |

**theory** *Wellfounded-More*

**imports** *Main*

**begin**

# 1 Transitions

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

## 1.1 More theorems about Closures

This is the equivalent of  $?r \leq ?s \implies ?r^{**} \leq ?s^{**}$  for *trancpl*

**lemma** *trancpl-mono-explicit*:

$r^{++} a b \implies r \leq s \implies s^{++} a b$

**using** *rtrancpl-mono* **by** (*auto dest!*: *trancplD intro: rtrancpl-into-trancpl2*)

**lemma** *trancpl-mono*:

**assumes** *mono*:  $r \leq s$

**shows**  $r^{++} \leq s^{++}$

**using** *rtrancpl-mono[OF mono]* *mono* **by** (*auto dest!*: *trancplD intro: rtrancpl-into-trancpl2*)

**lemma** *trancpl-idemp-rel*:

$R^{++++} a b \longleftrightarrow R^{++} a b$

**apply** (*rule iffI*)

**prefer** 2 **apply** *blast*

**by** (*induction rule: trancpl-induct*) *auto*

Equivalent of  $?r^{****} = ?r^{**}$

**lemma** *trancpl-idemp*:  $(r^+)^+ = r^+$

**by** *simp*

**lemmas** *trancpl-idemp[simp]* = *trancpl-idemp[to-pred]*

This theorem already exists as  $?r^{**} ?a ?b \equiv ?a = ?b \vee ?r^{++} ?a ?b$  (and sledgehammer uses it), but it makes sense to duplicate it, because it is unclear how stable the lemmas in Nitpick are.

**lemma** *rtrancpl-unfold*:  $rtrancpl r a b \longleftrightarrow (a = b \vee trancpl r a b)$

**by** (*meson rtrancpl.simps rtrancplD trancpl-into-rtrancpl*)

**lemma** *trancpl-unfold-end*:  $trancpl r a b \longleftrightarrow (\exists a'. rtrancpl r a a' \wedge r a' b)$

**by** (*metis rtrancpl.rtrancpl-refl rtrancpl-into-trancpl1 trancpl.cases trancpl-into-rtrancpl*)

**lemma** *trancpl-unfold-begin*:  $trancpl r a b \longleftrightarrow (\exists a'. r a a' \wedge rtrancpl r a' b)$

**by** (*meson rtrancpl-into-trancpl2 trancplD*)

**lemma** *trancpl-set-trancpl*:  $(a, b) \in \{(b, a). P a b\}^+ \longleftrightarrow P^{++} b a$

**apply** (*rule iffI*)

**apply** (*induction rule: trancpl-induct; simp*)

**apply** (*induction rule: trancpl-induct; auto simp: trancpl-into-trancpl2*)

**done**

**lemma** *trancpl-rtrancpl-rtrancpl-rel*:  $R^{++++} a b \longleftrightarrow R^{**} a b$

**by** (*simp add: rtrancpl-unfold*)

**lemma** *trancpl-rtrancpl-rtrancpl[simp]*:  $R^{++++} = R^{**}$

**by** (*fastforce simp: rtrancpl-unfold*)

```

lemma rtranclp-exists-last-with-prop:
  assumes  $R\ x\ z$ 
  and  $R^{**}\ z\ z'$  and  $P\ x\ z$ 
  shows  $\exists y\ y'.\ R^{**}\ x\ y \wedge R\ y\ y' \wedge P\ y\ y' \wedge (\lambda a\ b.\ R\ a\ b \wedge \neg P\ a\ b)^{**}\ y'\ z'$ 
  using assms(2,1,3)
proof (induction arbitrary: )
  case base
  then show ?case by auto
next
  case (step  $z'\ z''$ ) note  $z = \text{this}(2)$  and  $IH = \text{this}(3)[OF\ \text{this}(4-5)]$ 
  show ?case
  apply (cases  $P\ z'\ z''$ )
  apply (rule exI[of -  $z'$ ], rule exI[of -  $z''$ ])
  using  $z\ \text{assms}(1)\ \text{step.hyps}(1)\ \text{step.prem}(2)$  apply auto[1]
  using  $IH\ z\ \text{rtranclp.rtrancl-into-rtrancl}$  by fastforce
qed

```

```

lemma rtranclp-and-rtranclp-left:  $(\lambda a\ b.\ P\ a\ b \wedge Q\ a\ b)^{**}\ S\ T \implies P^{**}\ S\ T$ 
by (induction rule: rtranclp-induct) auto

```

## 1.2 Full Transitions

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

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

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

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

**lemma** *rtranclp-full1I*:  
 $R^{**}\ a\ b \implies \text{full1}\ R\ b\ c \implies \text{full1}\ R\ a\ c$   
**unfolding** *full1-def* **by** *auto*

**lemma** *trancpl-full1I*:  
 $R^{++}\ a\ b \implies \text{full1}\ R\ b\ c \implies \text{full1}\ R\ a\ c$   
**unfolding** *full1-def* **by** *auto*

**lemma** *rtranclp-fullI*:  
 $R^{**}\ a\ b \implies \text{full}\ R\ b\ c \implies \text{full}\ R\ a\ c$   
**unfolding** *full-def* **by** *auto*

**lemma** *trancpl-full-full1I*:  
 $R^{++}\ a\ b \implies \text{full}\ R\ b\ c \implies \text{full1}\ R\ a\ c$   
**unfolding** *full-def full1-def* **by** *auto*

**lemma** *full-fullI*:  
 $R\ a\ b \implies \text{full}\ R\ b\ c \implies \text{full1}\ R\ a\ c$   
**unfolding** *full-def full1-def* **by** *auto*

**lemma** *full-unfold*:  
 $\text{full}\ r\ S\ S' \longleftrightarrow ((S = S' \wedge \text{no-step}\ r\ S') \vee \text{full1}\ r\ S\ S')$   
**unfolding** *full-def full1-def* **by** (*auto simp add: rtranclp-unfold*)

**lemma** *full1-is-full[intro]*:  $full1\ R\ S\ T \implies full\ R\ S\ T$   
 by (*simp add: full-unfold*)

**lemma** *not-full1-rtranclp-relation*:  $\neg full1\ R^{**}\ a\ b$   
 by (*meson full1-def rtranclp.rtrancl-refl*)

**lemma** *not-full-rtranclp-relation*:  $\neg full\ R^{**}\ a\ b$   
 by (*meson full-full1 not-full1-rtranclp-relation rtranclp.rtrancl-refl*)

**lemma** *full1-tranclp-relation-full*:  
 $full1\ R^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b$   
 by (*metis converse-tranclpE full1-def reflclp-tranclp rtranclpD rtranclp-idemp rtranclp-reflclp tranclp.r-into-trancl tranclp-into-rtranclp*)

**lemma** *full-tranclp-relation-full*:  
 $full\ R^{++}\ a\ b \longleftrightarrow full\ R\ a\ b$   
 by (*metis full-unfold full1-tranclp-relation-full tranclp.r-into-trancl tranclpD*)

**lemma** *rtranclp-full1-eq-or-full1*:  
 $(full1\ R)^{**}\ a\ b \longleftrightarrow (a = b \vee full1\ R\ a\ b)$   
**proof** –  
 have  $\forall p\ a\ aa.\ \neg p^{**}\ (a::'a)\ aa \vee a = aa \vee (\exists ab.\ p^{**}\ a\ ab \wedge p\ ab\ aa)$   
 by (*metis rtranclp.cases*)  
 then obtain  $aa :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$  **where**  
 $f1: \forall p\ a\ ab.\ \neg p^{**}\ a\ ab \vee a = ab \vee p^{**}\ a\ (aa\ p\ a\ ab) \wedge p\ (aa\ p\ a\ ab)\ ab$   
 by *moura*  
 { **assume**  $a \neq b$   
 { **assume**  $\neg full1\ R\ a\ b \wedge a \neq b$   
 then have  $a \neq b \wedge a \neq b \wedge \neg full1\ R\ (aa\ (full1\ R)\ a\ b)\ b \vee \neg (full1\ R)^{**}\ a\ b \wedge a \neq b$   
 using  $f1$  by (*metis (no-types) full1-def full1-tranclp-relation-full*)  
 then have *?thesis*  
 using  $f1$  by *blast* }  
 then have *?thesis*  
 by *auto* }  
 then show *?thesis*  
 by *fastforce*  
**qed**

**lemma** *tranclp-full1-full1*:  
 $(full1\ R)^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b$   
 by (*metis full1-def rtranclp-full1-eq-or-full1 tranclp-unfold-begin*)

### 1.3 Well-Foundedness and Full Transitions

**lemma** *wf-exists-normal-form*:  
**assumes**  $wf: wf\ \{(x, y).\ R\ y\ x\}$   
**shows**  $\exists b.\ R^{**}\ a\ b \wedge no\text{-}step\ R\ b$   
**proof** (*rule ccontr*)  
**assume**  $\neg ?thesis$   
 then have  $H: \bigwedge b.\ \neg R^{**}\ a\ b \vee \neg no\text{-}step\ R\ b$   
 by *blast*  
 def  $F \equiv rec\text{-}nat\ a\ (\lambda i\ b.\ SOME\ c.\ R\ b\ c)$   
 have [*simp*]:  $F\ 0 = a$   
 unfolding *F-def* by *auto*  
 have [*simp*]:  $\bigwedge i.\ F\ (Suc\ i) = (SOME\ b.\ R\ (F\ i)\ b)$   
 using *F-def* by *simp*

```

{ fix i
  have  $\forall j < i. R (F j) (F (Suc j))$ 
  proof (induction i)
    case 0
    then show ?case by auto
  next
    case (Suc i)
    then have  $R^{**} a (F i)$ 
    by (induction i) auto
    then have  $R (F i) (SOME b. R (F i) b)$ 
    using H by (simp add: someI-ex)
    then have  $\forall j < Suc i. R (F j) (F (Suc j))$ 
    using H Suc by (simp add: less-Suc-eq)
    then show ?case by fast
  qed
}
then have  $\forall j. R (F j) (F (Suc j))$  by blast
then show False
  using wf unfolding wfP-def wf-iff-no-infinite-down-chain by blast
qed

```

```

lemma wf-exists-normal-form-full:
  assumes wf:wf  $\{(x, y). R y x\}$ 
  shows  $\exists b. full R a b$ 
  using wf-exists-normal-form[OF assms] unfolding full-def by blast

```

## 1.4 More Well-Foundedness

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

- link between  $wf$  and infinite chains:  $wf ?r = (\neg (\exists f. \forall i. (f (Suc i), f i) \in ?r)), \llbracket wf ?r; \bigwedge k. (?f (Suc k), ?f k) \notin ?r \implies ?thesis \rrbracket \implies ?thesis$

```

lemma wf-if-measure-in-wf:
  wf R  $\implies (\bigwedge a b. (a, b) \in S \implies (\nu a, \nu b) \in R) \implies wf S$ 
  by (metis in-inv-image wfE-min wfI-min wf-inv-image)

```

```

lemma wfP-if-measure: fixes f :: 'a  $\Rightarrow$  nat
shows  $(\bigwedge x y. P x \implies g x y \implies f y < f x) \implies wf \{(y, x). P x \wedge g x y\}$ 
  apply (insert wf-measure[of f])
  apply (simp only: measure-def inv-image-def less-than-def less-eq)
  apply (erule wf-subset)
  apply auto
done

```

```

lemma wf-if-measure-f:
  assumes wf r
  shows wf  $\{(b, a). (f b, f a) \in r\}$ 
  using assms by (metis inv-image-def wf-inv-image)

```

```

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  $\{(y, x). P x \wedge g x y\}$ 
  proof -
    have wf  $\{(b, a). (f b, f a) \in r\}$  using assms(1) wf-if-measure-f by auto

```



then have  $wf \{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\}$   
 using  $wf\text{-subset}[of - \{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\}]$  **by** *auto*  
 moreover have  $\{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\} \subseteq \{(b, a). (f b, f a) \in r\}$  **by** *auto*  
 moreover have  $\{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\} = \{(b, a). P a \wedge g a b\}$  **using** *H* **by** *auto*  
 ultimately show *?thesis* **using**  $wf\text{-subset}$  **by** *simp*  
**qed**

**lemma** *wf-lex-less*:  $wf (lex \{(a, b). (a::nat) < b\})$   
**proof** –  
 have  $m: \{(a, b). a < b\} = \text{measure id}$  **by** *auto*  
 show *?thesis* **apply** (*rule wf-lex*) **unfolding** *m* **by** *auto*  
**qed**

**lemma** *wfP-if-measure2*: **fixes**  $f :: 'a \Rightarrow nat$   
**shows**  $(\bigwedge x y. P x y \Longrightarrow g x y \Longrightarrow f x < f y) \Longrightarrow wf \{(x, y). P x y \wedge g x y\}$   
**apply**(*insert wf-measure[of f]*)  
**apply**(*simp only: measure-def inv-image-def less-than-def less-eq*)  
**apply**(*erule wf-subset*)  
**apply** *auto*  
**done**

**lemma** *lexord-on-finite-set-is-wf*:  
**assumes**  
*P-finite*:  $\bigwedge U. P U \longrightarrow U \in A$  **and**  
*finite*: *finite* *A* **and**  
*wf*:  $wf R$  **and**  
*trans*: *trans* *R*  
**shows**  $wf \{(T, S). (P S \wedge P T) \wedge (T, S) \in lexord R\}$   
**proof** (*rule wfP-if-measure2*)  
**fix** *T S*  
**assume** *P*:  $P S \wedge P T$  **and**  
*s-le-t*:  $(T, S) \in lexord R$   
**let** *?f* =  $\lambda S. \{U. (U, S) \in lexord R \wedge P U \wedge P S\}$   
**have**  $?f T \subseteq ?f S$   
 using *s-le-t* *P* *lexord-trans* *trans* **by** *auto*  
**moreover** **have**  $T \in ?f S$   
 using *s-le-t* *P* **by** *auto*  
**moreover** **have**  $T \notin ?f T$   
 using *s-le-t* **by** (*auto simp add: lexord-irreflexive local.wf*)  
**ultimately** **have**  $\{U. (U, T) \in lexord R \wedge P U \wedge P T\} \subset \{U. (U, S) \in lexord R \wedge P U \wedge P S\}$   
**by** *auto*  
**moreover** **have** *finite*  $\{U. (U, S) \in lexord R \wedge P U \wedge P S\}$   
 using *finite* **by** (*metis (no-types, lifting) P-finite finite-subset mem-Collect-eq subsetI*)  
**ultimately** **show**  $\text{card } (?f T) < \text{card } (?f S)$  **by** (*simp add: psubset-card-mono*)  
**qed**

**lemma** *wf-fst-wf-pair*:  
**assumes**  $wf \{(M', M). R M' M\}$   
**shows**  $wf \{((M', N'), (M, N)). R M' M\}$   
**proof** –  
**have**  $wf (\{(M', M). R M' M\} <*\text{lex}*> \{\})$   
 using *assms* **by** *auto*  
**then show** *?thesis*  
**by** (*rule wf-subset*) *auto*

qed

lemma *wf-snd-wf-pair*:

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

proof –

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

using *assms wf-fst-wf-pair* by *auto*

then have *wf*:  $\bigwedge P. (\forall x. (\forall y. (y, x) \in \{((M', N'), M, N). R M' M\} \longrightarrow P y) \longrightarrow P x) \implies \text{All } P$   
 unfolding *wf-def* by *auto*

show *?thesis*

unfolding *wf-def*

proof (*intro allI impI*)

fix *P* :: '*c* × '*a* ⇒ bool and *x* :: '*c* × '*a*

assume *H*:  $\forall x. (\forall y. (y, x) \in \{((M', N'), M, y). R N' y\} \longrightarrow P y) \longrightarrow P x$

obtain *a b* where *x*: *x* = (*a*, *b*) by (*cases x*)

have *P*: *P* *x* = (*P* ○ ( $\lambda(a, b). (b, a)$ )) (*b*, *a*)

unfolding *x* by *auto*

show *P* *x*

using *wf*[*of P* ○ ( $\lambda(a, b). (b, a)$ )] apply *rule*

using *H* apply *simp*

unfolding *P* by *blast*

qed

qed

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

assumes *wf* *r*

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

apply (*rule wf-subset*)

using *wf-if-measure-f*[*OF assms, of f*] by *auto*

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*  $\{(y, h x)| y x. P x \wedge g x y\}$

proof –

have *wf*  $\{(b, h a)|b a. (f b, f (h a)) \in r\}$  using *assms(1) wf-if-measure-f-notation2* by *auto*

then have *wf*  $\{(b, h a)|b a. P a \wedge g a b \wedge (f b, f (h a)) \in r\}$

using *wf-subset*[*of* -  $\{(b, h a)|b a. P a \wedge g a b \wedge (f b, f (h a)) \in r\}$ ] by *auto*

moreover have  $\{(b, h a)|b a. P a \wedge g a b \wedge (f b, f (h a)) \in r\}$

$\subseteq \{(b, h a)|b a. (f b, f (h a)) \in r\}$  by *auto*

moreover have  $\{(b, h a)|b a. P a \wedge g a b \wedge (f b, f (h a)) \in r\} = \{(b, h a)|b a. P a \wedge g a b\}$

using *H* by *auto*

ultimately show *?thesis* using *wf-subset* by *simp*

qed

end

theory *List-More*

imports *Main*

begin

## 2 Various Lemmas

Close to  $(\bigwedge n. \forall m < n. ?P m \implies ?P n) \implies ?P ?n$ , but with a separation between zero and non-zero, and case names.

```

thm nat-less-induct
lemma nat-less-induct-case[case-names 0 Suc]:
  assumes
     $P\ 0$  and
     $\bigwedge n. (\forall m < \text{Suc } n. P\ m) \implies P\ (\text{Suc } n)$ 
  shows  $P\ n$ 
  apply (induction rule: nat-less-induct)
  by (case-tac n) (auto intro: assms)

```

Bounded function have not 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  $f :: \text{nat} \Rightarrow \text{nat}$ 
  assumes unbound: unbounded  $f$ 
  shows  $\exists n. f\ n > m \wedge n > n_0$ 
proof (rule ccontr)
  assume  $H: \neg$  ?thesis
  have finite  $\{f\ n \mid n. n \leq n_0\}$ 
  by auto
  have  $\bigwedge n. f\ n \leq \text{Max } (\{f\ n \mid n. n \leq n_0\} \cup \{m\})$ 
  apply (case-tac n \leq n_0)
  apply (metis (mono-tags, lifting) Max-ge Un-insert-right \{finite \{f\ n \mid n. n \leq n_0\}\}
    finite-insert insertCI mem-Collect-eq sup-bot.right-neutral)
  by (metis (no-types, lifting) H Max-less-iff Un-insert-right \{finite \{f\ n \mid n. n \leq n_0\}\}
    finite-insert insertI1 insert-not-empty leI sup-bot.right-neutral)
  then show False
  using unbound unfolding bounded-def by auto
qed

```

```

lemma bounded-const-product:
  fixes  $k :: \text{nat}$  and  $f :: \text{nat} \Rightarrow \text{nat}$ 
  assumes  $k > 0$ 
  shows bounded  $f \longleftrightarrow$  bounded  $(\lambda i. k * f\ i)$ 
  unfolding bounded-def apply (rule iffI)
  using mult-le-mono2 apply blast
  by (meson assms le-less-trans less-or-eq-imp-le nat-mult-less-cancel-disj split-div-lemma)

```

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

```

lemma bounded-finite-linorder:
  fixes  $f :: 'a \Rightarrow 'a :: \{\text{finite}, \text{linorder}\}$ 
  shows bounded  $f$ 
proof –
  have  $\bigwedge x. f\ x \leq \text{Max } \{f\ x \mid x. \text{True}\}$ 
  by (metis (mono-tags) Max-ge finite mem-Collect-eq)
  then show ?thesis
  unfolding bounded-def by blast
qed

```

### 3 More List

#### 3.1 *upt*

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

**lemma** *upt-Suc-le-append*:  $\neg i \leq j \implies [i..<Suc\ j] = []$   
**by** *auto*

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

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

**lemma**  
**assumes**  $i \leq n - m$   
**shows**  $take\ i\ [m..<n] = [m..<m+i]$   
**by** (*metis Nat.le-diff-conv2 add commute assms diff-is-0-eq' linear take-upt upt-conv-Nil*)

The counterpart for this lemma when  $n - m < i$  is *length ?xs ≤ ?n ⇒ take ?n ?xs = ?xs*. It is close to  $?i + ?m \leq ?n \implies take\ ?m\ [?i..<?n] = [?i..<?i + ?m]$ , but seems more general.

**lemma** *take-upt-bound-minus[simp]*:  
**assumes**  $i \leq n - m$   
**shows**  $take\ i\ [m..<n] = [m..<m+i]$   
**using** *assms* **by** (*induction i*) *auto*

**lemma** *append-cons-eq-upt*:  
**assumes**  $A @ B = [m..<n]$   
**shows**  $A = [m..<m+length\ A]$  **and**  $B = [m + length\ A..<n]$   
**proof** –  
**have**  $take\ (length\ A)\ (A @ B) = A$  **by** *auto*  
**moreover**  
**have**  $length\ A \leq n - m$  **using** *assms linear calculation* **by** *fastforce*  
**then have**  $take\ (length\ A)\ [m..<n] = [m..<m+length\ A]$  **by** *auto*  
**ultimately show**  $A = [m..<m+length\ A]$  **using** *assms* **by** *auto*  
**show**  $B = [m + length\ A..<n]$  **using** *assms* **by** (*metis append-eq-conv-conj drop-upt*)  
**qed**

The converse of  $?A @ ?B = [?m..<?n] \implies ?A = [?m..<?m + length\ ?A]$

$?A @ ?B = [?m..<?n] \implies ?B = [?m + length\ ?A..<?n]$  does not hold, for example if  $B$  is empty and  $A$  is  $[0::'a]$ :

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

**oops**

A more restrictive version holds:

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

**proof**

**assume**  $?A$  **then show**  $?B$  **by** (*auto simp add: append-cons-eq-upt*)

**next**

**assume**  $?P$  **and**  $?B$

**then show**  $?A$  **using** *append-eq-conv-conj* **by** *fastforce*

qed

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

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

shows  $A = [m..<i]$

**proof** –

have  $A = [m..<m + \text{length } A]$  **using** *assms append-cons-eq-upt* **by** *auto*

have  $(A @ i \# B) ! (\text{length } A) = i$  **by** *auto*

moreover have  $n - m = \text{length } (A @ i \# B)$

using *assms length-upt* **by** *presburger*

then have  $[m..<n] ! (\text{length } A) = m + \text{length } A$  **by** *simp*

ultimately have  $i = m + \text{length } A$  **using** *assms* **by** *auto*

then show *?thesis* **using**  $\langle A = [m..<m + \text{length } A] \rangle$  **by** *auto*

qed

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

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

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

**using** *assms*

**proof** (*induction A arbitrary: m*)

case *Nil*

then show *?case* **by** (*metis append-Nil diff-is-0-eq list.size(3) order-refl upt-eq-Cons-conv*)

**next**

case (*Cons a A*)

then have  $A : A @ i \# B = [m + 1..<n]$  **by** (*metis append-Cons upt-eq-Cons-conv*)

then have  $m < i$  **by** (*metis Cons.premis append-cons-eq-upt-length-i upt-eq-Cons-conv*)

with *Cons.IH[OF A]* show *?case* **by** *auto*

qed

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

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

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

**proof** –

have  $B = [\text{Suc } m + \text{length } A..<n]$  **using** *assms append-cons-eq-upt[of A @ [i] B m n]* **by** *auto*

have  $(A @ i \# B) ! (\text{length } A) = i$  **by** *auto*

moreover have  $n - m = \text{length } (A @ i \# B)$

using *assms length-upt* **by** *auto*

then have  $[m..<n] ! (\text{length } A) = m + \text{length } A$  **by** *simp*

ultimately have  $i = m + \text{length } A$  **using** *assms* **by** *auto*

then show *?thesis* **using**  $\langle B = [\text{Suc } m + \text{length } A..<n] \rangle$  **by** *auto*

qed

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

**proof** (*induct n*)

case *0*

then show *?case* **by** *simp*

**next**

case (*Suc n*) **note** *IH = this*

have  $i : \text{insert } 0 \{ \text{Suc } 0..<\text{Suc } n \} = \text{insert } 0 \{ \text{Suc } 0..<n \} \cup \{n\}$  **by** *auto*

show *?case* **using** *IH* **unfolding** *i* **by** *auto*

qed

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

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

shows  $i < j$

**proof** –

**have**  $xs$ :  $xs = [m \dots i]$  **and**  $ys$ :  $ys = [Suc\ i \dots j]$  **and**  $zs$ :  $zs = [Suc\ j \dots n]$   
**using**  $H$  **by** (*auto dest: append-cons-eq-upt-length-i append-cons-eq-upt-length-i-end*)  
**show** *?thesis*  
**by** (*metis append-cons-eq-upt-length-i-end assms lessI less-trans self-append-conv2*  
*upt-eq-Cons-conv upt-rec ys*)

**qed**

### 3.2 Lexicographic ordering

We are working a lot on lexicographic ordering over pairs.

**lemma** *list-length2-append-cons*:

$[c, d] = ys @ y \# ys' \longleftrightarrow (ys = [] \wedge y = c \wedge ys' = [d]) \vee (ys = [c] \wedge y = d \wedge ys' = [])$   
**by** (*cases ys; cases ys'*) *auto*

**lemma** *lexn2-conv*:

$([a, b], [c, d]) \in \text{lexn}\ r\ 2 \longleftrightarrow (a, c) \in r \vee (a = c \wedge (b, d) \in r)$   
**unfolding** *lexn-conv* **by** (*auto simp add: list-length2-append-cons*)

**end**

**theory** *Prop-Logic*

**imports** *Main*

**begin**

## 4 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.

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

**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  $P\ \psi$ 
apply (induct rule: propo.induct)
using assms by metis+

```

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:

```

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  $P$ 
using assms by (case-tac c, auto simp add: binary-connectives-def)

```

**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  $P$ 
using assms by (case-tac c, auto simp add: binary-connectives-def)

```

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**  $P \ x$   
**using** *assms* **by** *induction* (*auto simp add: binary-connectives-def*)

## 4.2 properties of the abstraction

First we can define simplification rules.

**lemma** *wf-conn-conn[simp]*:  
 $wf\_conn \ CT \ l \implies conn \ CT \ l = FT$   
 $wf\_conn \ CF \ l \implies conn \ CF \ l = FF$   
 $wf\_conn \ (CVar \ x) \ l \implies conn \ (CVar \ x) \ l = FVar \ x$   
**apply** (*simp-all add: wf-conn.simps*)  
**unfolding** *binary-connectives-def* **by** *simp-all*

**lemma** *wf-conn-list-decomp[simp]*:  
 $wf\_conn \ CT \ l \longleftrightarrow l = []$   
 $wf\_conn \ CF \ l \longleftrightarrow l = []$   
 $wf\_conn \ (CVar \ x) \ l \longleftrightarrow l = []$   
 $wf\_conn \ CNot \ (\xi \ @ \ \varphi \ \# \ \xi') \longleftrightarrow \xi = [] \wedge \xi' = []$   
**apply** (*simp-all add: wf-conn.simps*)  
**unfolding** *binary-connectives-def* **apply** *simp-all*  
**by** (*metis append-Nil append-is-Nil-conv list.distinct(1) list.sel(3) tl-append2*)

**lemma** *wf-conn-list*:  
 $wf\_conn \ c \ l \implies conn \ c \ l = FT \longleftrightarrow (c = CT \wedge l = [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FF \longleftrightarrow (c = CF \wedge l = [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FVar \ x \longleftrightarrow (c = CVar \ x \wedge l = [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FAnd \ a \ b \longleftrightarrow (c = CAnd \wedge l = a \ \# \ b \ \# \ [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FOr \ a \ b \longleftrightarrow (c = COr \wedge l = a \ \# \ b \ \# \ [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FEq \ a \ b \longleftrightarrow (c = CEq \wedge l = a \ \# \ b \ \# \ [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FImp \ a \ b \longleftrightarrow (c = CImp \wedge l = a \ \# \ b \ \# \ [])$   
 $wf\_conn \ c \ l \implies conn \ c \ l = FNot \ a \longleftrightarrow (c = CNot \wedge l = a \ \# \ [])$   
**apply** (*induct l rule: wf-conn.induct*)  
**unfolding** *binary-connectives-def* **by** *auto*

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

**lemma** *list-length2-decomp*:  $length \ l = 2 \implies (\exists \ a \ b. \ l = a \ \# \ b \ \# \ [])$   
**apply** (*induct l, auto*)  
**by** (*case-tac l, auto*)

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

**lemma** *wf-conn-bin-list-length*:  
**fixes**  $l :: 'v \ \text{prop}$   
**assumes**  $conn: c \in \text{binary-connectives}$   
**shows**  $length \ l = 2 \longleftrightarrow wf\_conn \ c \ l$   
**proof**  
**assume**  $length \ l = 2$   
**thus**  $wf\_conn \ c \ l$  **using** *wf-conn-binary list-length2-decomp* **using** *conn* **by** *metis*  
**next**  
**assume**  $wf\_conn \ c \ l$   
**thus**  $length \ l = 2$  (*is ?P l*)



```

proof (cases rule: wf-conn.induct)
  case wf-conn-nullary
  thus ?P [] using conn binary-connectives-def
    using connective.distinct(11) connective.distinct(13) connective.distinct(9) by blast
next
  fix  $\psi :: 'v$  propo
  case wf-conn-unary
  thus ?P [ $\psi$ ] using conn binary-connectives-def
    using connective.distinct by blast
next
  fix  $\psi \ \psi' :: 'v$  propo
  show ?P [ $\psi, \psi'$ ] by auto
qed
qed

```

```

lemma wf-conn-not-list-length[iff]:
  fixes  $l :: 'v$  propo list
  shows wf-conn CNot  $l \longleftrightarrow \text{length } l = 1$ 
  apply auto
  apply (metis append-Nil connective.distinct(5,17,27) length-Cons list.size(3) wf-conn.simps
    wf-conn-list-decomp(4))
  by (simp add: length-Suc-conv wf-conn.simps)

```

Decomposing the Not into an element is moreover very useful.

```

lemma wf-conn-Not-decomp:
  fixes  $l :: 'v$  propo list and  $a :: 'v$ 
  assumes corr: wf-conn CNot  $l$ 
  shows  $\exists a. l = [a]$ 
  by (metis (no-types, lifting) One-nat-def Suc-length-conv corr length-0-conv wf-conn-not-list-length)

```

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:
   $\text{length } l = \text{length } l' \implies \text{wf-conn } c \ l \longleftrightarrow \text{wf-conn } c \ l'$ 
proof -
  {
    fix  $l \ l'$ 
    have  $\text{length } l = \text{length } l' \implies \text{wf-conn } c \ l \implies \text{wf-conn } c \ l'$ 
    apply (cases c l rule: wf-conn.induct, auto)
    by (metis wf-conn-bin-list-length)
  }
  thus  $\text{length } l = \text{length } l' \implies \text{wf-conn } c \ l = \text{wf-conn } c \ l'$  by metis
qed

```

```

lemma wf-conn-no-arity-change-helper:
   $\text{length } (\xi @ \varphi \# \xi') = \text{length } (\xi @ \varphi' \# \xi')$ 
  by auto

```

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:  $\text{conn } c \ l = \text{FNot } \psi$ 
  shows  $c = \text{CNot}$  and  $l = [\psi]$ 
  apply (cases c l rule: wf-conn.cases)
  using correct conn unfolding binary-connectives-def apply auto

```

```

apply (cases c l rule: wf-conn.cases)
using correct conn unfolding binary-connectives-def by auto

```

```

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
  using corr
proof (cases ca l rule: wf-conn.cases)
  case (wf-conn-nullary v)
  thus ca = c  $\wedge$   $\psi$ s = l using assms
    by (metis conn.simps(1) conn.simps(2) conn.simps(3) wf-conn-list(1-3))
next
  case (wf-conn-unary  $\psi'$ )
  hence *: FNot  $\psi'$  = conn c  $\psi$ s using conn-inj-not eq assms by auto
  hence c = ca by (metis conn-inj-not(1) corr' wf-conn-unary(2))
  moreover have  $\psi$ s = l using * conn-inj-not(2) corr' wf-conn-unary(1) by force
  ultimately show ca = c  $\wedge$   $\psi$ s = l by auto
next
  case (wf-conn-binary  $\psi'$   $\psi''$ )
  thus ca = c  $\wedge$   $\psi$ s = l
    using eq corr' unfolding binary-connectives-def apply (case-tac ca, auto simp add: wf-conn-list)
    using wf-conn-list(4-7) corr' by metis+
qed

```

### 4.3 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:  $\psi \in \text{set } l \implies \text{wf-conn } c \ l \implies \varphi \preceq \psi \implies \varphi \preceq \text{conn } c \ l$ 

```

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  $\varphi \preceq \psi \implies \varphi \preceq \psi$ 
  apply (induct rule: subformula.induct)
  using subformula-into-subformula wf-conn-unary subformula-refl list.set-intros(1) subformula-refl
  by (fastforce intro: subformula-into-subformula)+

```

```

lemma subformula-in-binary-conn:
  assumes conn: c  $\in$  binary-connectives
  shows f  $\preceq$  conn c [f, g]
  and g  $\preceq$  conn c [f, g]
proof -
  have a: wf-conn c (f# [g]) using conn wf-conn-binary binary-connectives-def by auto
  moreover have b: f  $\preceq$  f using subformula-refl by auto

```

ultimately show  $f \preceq \text{conn } c [f, g]$   
 by (metis append-Nil in-set-conv-decomp subformula-into-subformula)  
 next  
 have  $a: \text{wf-conn } c ([f] @ [g])$  using *conn wf-conn-binary binary-connectives-def* by auto  
 moreover have  $b: g \preceq g$  using *subformula-refl* by auto  
 ultimately show  $g \preceq \text{conn } c [f, g]$  using *subformula-into-subformula* by force  
 qed

**lemma** *subformula-trans*:

$\psi \preceq \psi' \implies \varphi \preceq \psi \implies \varphi \preceq \psi'$   
 apply (induct  $\psi'$  rule: *subformula.inducts*)  
 by (auto simp add: *subformula-into-subformula*)

**lemma** *subformula-leaf*:

fixes  $\varphi \psi :: 'v \text{ propo}$   
 assumes *incl*:  $\varphi \preceq \psi$   
 and *simple*:  $\psi = FT \vee \psi = FF \vee \psi = FVar x$   
 shows  $\varphi = \psi$   
 using *incl simple*  
 by (induct rule: *subformula.induct*, auto simp add: *wf-conn-list*)

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

assumes  $\varphi \preceq \text{conn } c l$   
 and *wf-conn*  $c l$   
 and  $\forall \psi. \psi \in \text{set } l \longrightarrow \neg \varphi \preceq \psi$   
 shows  $\varphi = \text{conn } c l$   
 using *assms* apply (induction *conn c l* rule: *subformula.induct*, auto)  
 using *conn-inj* by blast

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

$\text{wf-conn } c l \implies \varphi \preceq \text{conn } c l \longleftrightarrow (\varphi = \text{conn } c l \vee (\exists \psi. \psi \in \text{set } l \wedge \varphi \preceq \psi))$   
 apply *standard*  
 using *subformula-not-incl-eq* apply *metis*  
 by (auto simp add: *subformula-into-subformula*)

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

$\varphi \preceq FAnd \psi \psi' \longleftrightarrow (\varphi = FAnd \psi \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$  (is ?P FAnd)  
 $\varphi \preceq FOr \psi \psi' \longleftrightarrow (\varphi = FOr \psi \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$   
 $\varphi \preceq FEq \psi \psi' \longleftrightarrow (\varphi = FEq \psi \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$   
 $\varphi \preceq FImp \psi \psi' \longleftrightarrow (\varphi = FImp \psi \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$

**proof** –

have *wf-conn*  $CAnd [\psi, \psi']$  by (simp add: *binary-connectives-def*)  
 hence  $\varphi \preceq \text{conn } CAnd [\psi, \psi'] \longleftrightarrow (\varphi = \text{conn } CAnd [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi''))$   
 using *wf-subformula-conn-cases* by *metis*  
 thus ?P FAnd by auto

**next**

have *wf-conn*  $COr [\psi, \psi']$  by (simp add: *binary-connectives-def*)  
 hence  $\varphi \preceq \text{conn } COr [\psi, \psi'] \longleftrightarrow (\varphi = \text{conn } COr [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi''))$   
 using *wf-subformula-conn-cases* by *metis*  
 thus ?P FOr by auto

**next**

have *wf-conn*  $CEq [\psi, \psi']$  by (simp add: *binary-connectives-def*)  
 hence  $\varphi \preceq \text{conn } CEq [\psi, \psi'] \longleftrightarrow (\varphi = \text{conn } CEq [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi''))$

```

    using wf-subformula-conn-cases by metis
  thus ?P FEq by auto
next
  have wf-conn CImp  $[\psi, \psi']$  by (simp add: binary-connectives-def)
  hence  $\varphi \preceq \text{conn CImp } [\psi, \psi'] \longleftrightarrow (\varphi = \text{conn CImp } [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi''))$ 
    using wf-subformula-conn-cases by metis
  thus ?P FImp by auto
qed

```

```

lemma wf-conn-helper-facts[iff]:
  wf-conn CNot  $[\varphi]$ 
  wf-conn CT []
  wf-conn CF []
  wf-conn (CVar  $x$ ) []
  wf-conn CAnd  $[\varphi, \psi]$ 
  wf-conn COr  $[\varphi, \psi]$ 
  wf-conn CImp  $[\varphi, \psi]$ 
  wf-conn CEq  $[\varphi, \psi]$ 
  using wf-conn.intros unfolding binary-connectives-def by fastforce+

```

```

lemma exists-c-conn:  $\exists c l. \varphi = \text{conn } c l \wedge \text{wf-conn } c l$ 
  by (cases  $\varphi$ ) force+

```

```

lemma subformula-conn-decomp[simp]:
  wf-conn  $c l \implies \varphi \preceq \text{conn } c l \longleftrightarrow (\varphi = \text{conn } c l \vee (\exists \psi \in \text{set } l. \varphi \preceq \psi))$ 
  apply auto
proof -
  {
    fix  $\xi$ 
    have  $\varphi \preceq \xi \implies \xi = \text{conn } c l \implies \text{wf-conn } c l \implies \forall x::'a \text{ propo} \in \text{set } l. \neg \varphi \preceq x \implies \varphi = \text{conn } c l$ 
      apply (induct rule: subformula.induct)
      apply simp
      using conn-inj by blast
    }
  moreover assume wf-conn  $c l$  and  $\varphi \preceq \text{conn } c l$  and  $\forall x::'a \text{ propo} \in \text{set } l. \neg \varphi \preceq x$ 
  ultimately show  $\varphi = \text{conn } c l$  by metis
next
  fix  $\psi$ 
  assume wf-conn  $c l$  and  $\psi \in \text{set } l$  and  $\varphi \preceq \psi$ 
  thus  $\varphi \preceq \text{conn } c l$  using wf-subformula-conn-cases by blast
qed

```

```

lemma subformula-leaf-explicit[simp]:
   $\varphi \preceq FT \longleftrightarrow \varphi = FT$ 
   $\varphi \preceq FF \longleftrightarrow \varphi = FF$ 
   $\varphi \preceq FVar x \longleftrightarrow \varphi = FVar x$ 
  apply auto
  using subformula-leaf by metis +

```

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

```

primrec vars-of-prop:: ' $v$  propo  $\Rightarrow$  ' $v$  set where
  vars-of-prop FT = {} |
  vars-of-prop FF = {} |
  vars-of-prop (FVar  $x$ ) = { $x$ } |

```

$\text{vars-of-prop } (F\text{Not } \varphi) = \text{vars-of-prop } \varphi \mid$   
 $\text{vars-of-prop } (F\text{And } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi \mid$   
 $\text{vars-of-prop } (F\text{Or } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi \mid$   
 $\text{vars-of-prop } (F\text{Imp } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi \mid$   
 $\text{vars-of-prop } (F\text{Eq } \varphi \ \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi$

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

**fixes**  $\xi \ \xi' :: 'v \text{ propo list}$  **and**  $\psi :: 'v \text{ propo}$  **and**  $c :: 'v \text{ connective}$   
**assumes** *corr*:  $\text{wf-conn } c \ l$  **and** *incl*:  $\psi \in \text{set } l$   
**shows**  $\text{vars-of-prop } \psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$

**proof** (*cases c rule: connective-cases-arity-2*)

**case** *nullary*

**hence** *False* **using** *corr incl* **by** *auto*

**thus**  $\text{vars-of-prop } \psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$  **by** *blast*

**next**

**case** *binary* **note**  $c = \text{this}$

**then obtain**  $a \ b$  **where**  $ab: l = [a, b]$

**using** *wf-conn-bin-list-length list-length2-decomp corr* **by** *metis*

**hence**  $\psi = a \vee \psi = b$  **using** *incl* **by** *auto*

**thus**  $\text{vars-of-prop } \psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$

**using**  $ab \ c$  **unfolding** *binary-connectives-def* **by** *auto*

**next**

**case** *unary* **note**  $c = \text{this}$

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

**have**  $l = [\psi]$  **using** *corr c incl split-list* **by** *force*

**thus**  $\text{vars-of-prop } \psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$  **using**  $c$  **by** *auto*

**qed**

The set of variables is compatible with the subformula order.

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

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

**apply** (*induct rule: subformula.induct*)

**apply** *simp*

**using** *vars-of-prop-incl-conn* **by** *blast*

## 4.4 Positions

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

**datatype** *sign* =  $L \mid R$

We use *nil* instead of  $\varepsilon$ .

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

*pos FF* =  $\{\{\}\}$  **|**

*pos FT* =  $\{\{\}\}$  **|**

*pos (FVar x)* =  $\{\{\}\}$  **|**

*pos (FAnd  $\varphi \ \psi$ )* =  $\{\{\}\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\}$  **|**

*pos (FOr  $\varphi \ \psi$ )* =  $\{\{\}\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\}$  **|**

*pos (FEq  $\varphi \ \psi$ )* =  $\{\{\}\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\}$  **|**

*pos (FImp  $\varphi \ \psi$ )* =  $\{\{\}\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\}$  **|**

*pos (FNot  $\varphi$ )* =  $\{\{\}\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\}$

**lemma** *finite-pos*: *finite* (*pos*  $\varphi$ )

**by** (*induct*  $\varphi$ , *auto*)

**lemma** *finite-inj-comp-set*:  
**fixes**  $s :: 'v \text{ set}$   
**assumes** *finite*:  $\text{finite } s$   
**and** *inj*:  $\text{inj } f$   
**shows**  $\text{card } (\{f \ p \mid p. p \in s\}) = \text{card } s$   
**using** *finite*  
**proof** (*induct s rule: finite-induct*)  
**show**  $\text{card } \{f \ p \mid p. p \in \{\}\} = \text{card } \{\}$  **by** *auto*  
**next**  
**fix**  $x :: 'v$  **and**  $s :: 'v \text{ set}$   
**assume** *f*:  $\text{finite } s$  **and** *notin*:  $x \notin s$   
**and** *IH*:  $\text{card } \{f \ p \mid p. p \in s\} = \text{card } s$   
**have** *f'*:  $\text{finite } \{f \ p \mid p. p \in \text{insert } x \ s\}$  **using** *f* **by** *auto*  
**have** *notin'*:  $f \ x \notin \{f \ p \mid p. p \in s\}$  **using** *notin inj injD* **by** *fastforce*  
**have**  $\{f \ p \mid p. p \in \text{insert } x \ s\} = \text{insert } (f \ x) \ \{f \ p \mid p. p \in s\}$  **by** *auto*  
**hence**  $\text{card } \{f \ p \mid p. p \in \text{insert } x \ s\} = 1 + \text{card } \{f \ p \mid p. p \in s\}$   
**using** *finite card-insert-disjoint f' notin'* **by** *auto*  
**moreover** **have**  $\dots = \text{card } (\text{insert } x \ s)$  **using** *notin f IH* **by** *auto*  
**finally** **show**  $\text{card } \{f \ p \mid p. p \in \text{insert } x \ s\} = \text{card } (\text{insert } x \ s)$  .  
**qed**

**lemma** *cons-inject*:  
 $\text{inj } (op \ \# \ s)$   
**by** (*meson injI list.inject*)

**lemma** *finite-insert-nil-cons*:  
 $\text{finite } s \implies \text{card } (\text{insert } [] \ \{L \ \# \ p \mid p. p \in s\}) = 1 + \text{card } \{L \ \# \ p \mid p. p \in s\}$   
**using** *card-insert-disjoint* **by** *auto*

**lemma** *cord-not[simp]*:  
 $\text{card } (pos \ (FNot \ \varphi)) = 1 + \text{card } (pos \ \varphi)$   
**by** (*simp add: cons-inject finite-inj-comp-set finite-pos*)

**lemma** *card-seperate*:  
**assumes** *finite s1* **and** *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**  $\text{card } (?L \cup ?R) = \text{card } ?L + \text{card } ?R$ )  
**proof** –  
**have** *finite ?L* **using** *assms* **by** *auto*  
**moreover** **have** *finite ?R* **using** *assms* **by** *auto*  
**moreover** **have**  $?L \cap ?R = \{\}$  **by** *blast*  
**ultimately** **show** *?thesis* **using** *assms card-Un-disjoint* **by** *blast*  
**qed**

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

**lemma** *prop-size-vars-of-prop*:  
**fixes**  $\varphi :: 'v \text{ propo}$   
**shows**  $\text{card } (\text{vars-of-prop } \varphi) \leq \text{prop-size } \varphi$   
**unfolding** *prop-size-def* **apply** (*induct*  $\varphi$ , *auto simp add: cons-inject finite-inj-comp-set finite-pos*)

```

proof –
  fix  $\varphi 1 \ \varphi 2 :: 'v \text{ propo}$ 
  assume  $IH1: \text{card} (\text{vars-of-prop } \varphi 1) \leq \text{card} (\text{pos } \varphi 1)$ 
  and  $IH2: \text{card} (\text{vars-of-prop } \varphi 2) \leq \text{card} (\text{pos } \varphi 2)$ 
  let  $?L = \{L \# p \mid p. p \in \text{pos } \varphi 1\}$ 
  let  $?R = \{R \# p \mid p. p \in \text{pos } \varphi 2\}$ 
  have  $\text{card} (?L \cup ?R) = \text{card } ?L + \text{card } ?R$ 
    using card-seperate finite-pos by blast
  moreover have  $\dots = \text{card} (\text{pos } \varphi 1) + \text{card} (\text{pos } \varphi 2)$ 
    by (simp add: cons-inject finite-inj-comp-set finite-pos)
  moreover have  $\dots \geq \text{card} (\text{vars-of-prop } \varphi 1) + \text{card} (\text{vars-of-prop } \varphi 2)$  using  $IH1 \ IH2$  by arith
  hence  $\dots \geq \text{card} (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2)$  using card-Un-le le-trans by blast
  ultimately
    show  $\text{card} (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc} (\text{card} (?L \cup ?R))$ 
       $\text{card} (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc} (\text{card} (?L \cup ?R))$ 
       $\text{card} (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc} (\text{card} (?L \cup ?R))$ 
       $\text{card} (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc} (\text{card} (?L \cup ?R))$ 
    by auto
qed

```

```

value  $\text{pos} (FImp (FAnd (FVar P) (FVar Q)) (FOr (FVar P) (FVar Q)))$ 

```

```

inductive  $\text{path-to} :: \text{sign list} \Rightarrow 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  where
   $\text{path-to-refl[intro]: path-to [] } \varphi \ \varphi \mid$ 
   $\text{path-to-l: } c \in \text{binary-connectives} \vee c = CNot \implies \text{wf-conn } c (\varphi \# l) \implies \text{path-to } p \ \varphi \ \varphi'$ 
     $\implies \text{path-to } (L \# p) (\text{conn } c (\varphi \# l)) \ \varphi' \mid$ 
   $\text{path-to-r: } c \in \text{binary-connectives} \implies \text{wf-conn } c (\psi \# \varphi \# []) \implies \text{path-to } p \ \varphi \ \varphi'$ 
     $\implies \text{path-to } (R \# p) (\text{conn } c (\psi \# \varphi \# [])) \ \varphi'$ 

```

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  $\text{path-to-subformula:}$ 
   $\text{path-to } p \ \varphi \ \varphi' \implies \varphi' \preceq \varphi$ 
  apply (induct rule: path-to.induct)
  apply simp
  apply (metis list.set-intros(1) subformula-into-subformula)
  using subformula-trans subformula-in-binary-conn(2) by metis

```

```

lemma  $\text{subformula-path-exists:}$ 
  fixes  $\varphi \ \varphi' :: 'v \text{ propo}$ 
  shows  $\varphi' \preceq \varphi \implies \exists p. \text{path-to } p \ \varphi \ \varphi'$ 
proof (induct rule: subformula.induct)
  case subformula-refl
  have  $\text{path-to [] } \varphi' \ \varphi'$  by auto
  thus  $\exists p. \text{path-to } p \ \varphi' \ \varphi'$  by metis

```

```

next
  case (subformula-into-subformula  $\psi \ l \ c$ )
  note  $\text{wf} = \text{this}(2)$  and  $IH = \text{this}(4)$  and  $\psi = \text{this}(1)$ 
  then obtain  $p$  where  $p: \text{path-to } p \ \psi \ \varphi'$  by metis
  {
    fix  $x :: 'v$ 
    assume  $c = CT \vee c = CF \vee c = CVar \ x$ 
    hence False using subformula-into-subformula by auto
  }

```

```

  hence  $\exists p. \text{path-to } p \text{ (conn } c \text{ l) } \varphi'$  by blast
}
moreover {
  assume  $c: c = CNot$ 
  hence  $l = [\psi]$  using wf  $\psi$  wf-conn-Not-decomp by fastforce
  hence  $\text{path-to } (L \# p) \text{ (conn } c \text{ l) } \varphi'$  by (metis c wf-conn-unary p path-to-l)
  hence  $\exists p. \text{path-to } p \text{ (conn } c \text{ l) } \varphi'$  by blast
}
moreover {
  assume  $c: c \in \text{binary-connectives}$ 
  obtain  $a \ b$  where  $ab: [a, b] = l$  using subformula-into-subformula c wf-conn-bin-list-length
  list-length2-decomp by metis
  hence  $a = \psi \vee b = \psi$  using  $\psi$  by auto
  hence  $\text{path-to } (L \# p) \text{ (conn } c \text{ l) } \varphi' \vee \text{path-to } (R \# p) \text{ (conn } c \text{ l) } \varphi'$  using  $c$  path-to-l
  path-to-r p ab by (metis wf-conn-binary)
  hence  $\exists p. \text{path-to } p \text{ (conn } c \text{ l) } \varphi'$  by blast
}
ultimately show  $\exists p. \text{path-to } p \text{ (conn } c \text{ l) } \varphi'$  using connective-cases-arity by metis
qed

```

```

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

```

## 5 Semantics over the syntax

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

```

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

```

```

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.

**lemma** *deduction-rule*:

$(\varphi \models_f \psi) \longleftrightarrow (\forall A. (A \models FImp \ \varphi \ \psi))$

**proof**

**assume**  $H: \varphi \models_f \psi$



```
{
  fix A
```

“Suppose that  $\varphi$  entails  $\psi$  (assumption  $\varphi \models^f \psi$ ) and let  $A$  be an arbitrary  $'v$ -valuation. We need to show  $A \models FImp \varphi \psi$ . ”

```
{
```

If  $A \varphi = (1::'b)$ , then  $A \varphi = (1::'b)$ , because  $\varphi$  entails  $\psi$ , and therefore  $A \models FImp \varphi \psi$ .

```
  assume A ⊨ φ
  hence A ⊨ ψ using H unfolding evalf-def by metis
  hence A ⊨ FImp φ ψ by auto
}
```

```
  moreover {
```

For otherwise, if  $A \varphi = (0::'b)$ , then  $A \models FImp \varphi \psi$  holds by definition, independently of the value of  $A \models \psi$ .

```
  assume ¬ A ⊨ φ
  hence A ⊨ FImp φ ψ by auto
}
```

In both cases  $A \models FImp \varphi \psi$ .

```
  ultimately have A ⊨ FImp φ ψ by blast
}
```

```
thus ∀ A. A ⊨ FImp φ ψ by blast
```

```
next
```

```
show ∀ A. A ⊨ FImp φ ψ ⟹ φ ⊨f ψ
```

```
proof (rule ccontr)
```

```
  assume ¬φ ⊨f ψ
```

```
  then obtain A where A ⊨ φ ∧ ¬A ⊨ ψ using evalf-def by metis
```

```
  hence ¬ A ⊨ FImp φ ψ by auto
```

```
  moreover assume ∀ A. A ⊨ FImp φ ψ
```

```
  ultimately show False by blast
```

```
qed
```

```
qed
```

A shorter proof:

```
lemma φ ⊨f ψ ⟷ (∀ A. A ⊨ FImp φ ψ)
```

```
by (simp add: evalf-def)
```

**definition** *same-over-set::* ( $'v \Rightarrow bool$ )  $\Rightarrow$  ( $'v \Rightarrow bool$ )  $\Rightarrow$   $'v$  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 φ)
```

```
shows A ⊨ φ ⟷ B ⊨ φ
```

```
using assms unfolding same-over-set-def by (induct φ, auto)
```

```
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.

## 6 Rewrite systems and properties

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

Here is a more precise link between the lifting and the subformulas: if a rewriting takes place between  $\varphi$  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** *propo-rew-step* r  $\varphi$   $\varphi' \implies \exists \psi \psi'. \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r \psi \psi'$

**apply** (*induct* rule: *propo-rew-step.induct*)

**using** *subformula.simps* *subformula-into-subformula* **apply** *blast*

**using** *wf-conn-no-arity-change* *subformula-into-subformula* *wf-conn-no-arity-change-helper*  
*in-set-conv-decomp* **by** *metis*

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 propo

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

**proof** (*induct*  $\varphi$  rule: *subformula.induct*)

**case** *subformula-refl*

**hence** *propo-rew-step* r  $\psi \psi'$  **using** *propo-rew-step.intros* **by** *auto*

**moreover** **have**  $\psi' \preceq \psi'$  **using** *Prop-Logic.subformula-refl* **by** *auto*

**ultimately show**  $\exists \varphi'. \psi' \preceq \varphi' \wedge \text{propo-rew-step } r \varphi \varphi'$  **by** *fastforce*

**next**

**case** (*subformula-into-subformula*  $\psi''$  l c)

**note**  $IH = \text{this}(4)$  **and**  $r = \text{this}(5)$  **and**  $\psi'' = \text{this}(1)$  **and**  $wf = \text{this}(2)$  **and**  $incl = \text{this}(3)$

**then obtain**  $\varphi'$  **where**  $\psi' \preceq \varphi' \wedge \text{propo-rew-step } r \psi'' \varphi'$  **by** *metis*

**moreover obtain**  $\xi \xi' ::$  'v propo list **where**

$l: l = \xi @ \psi'' \# \xi'$  **using** *List.split-list*  $\psi''$  **by** *metis*

**ultimately have** *propo-rew-step* r (conn c l) (conn c ( $\xi @ \varphi' \# \xi'$ ))

**using** *propo-rew-step.intros*(2) *wf* **by** *metis*

**moreover have**  $\psi' \preceq \text{conn } c (\xi @ \varphi' \# \xi')$

**using**  $wf * wf\text{-}conn\text{-}no\text{-}arity\text{-}change$  *Prop-Logic.subformula-into-subformula*

**by** (*metis* (*no-types*) *in-set-conv-decomp* l *wf-conn-no-arity-change-helper*)

**ultimately show**  $\exists \varphi'. \psi' \preceq \varphi' \wedge \text{propo-rew-step } r (\text{conn } c l) \varphi'$  **by** *metis*

**qed**

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

$(\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi') \longleftrightarrow (\exists \varphi'. \text{propo-rew-step } r \varphi \varphi')$

```

using propo-rew-step-subformula-imp propo-rew-step-subformula-rec by metis+

lemma consistency-decompose-into-list:
  assumes wf: wf-conn c l and wf': wf-conn c l'
  and same:  $\forall n. (A \models l ! n \longleftrightarrow (A \models l' ! n))$ 
  shows  $(A \models \text{conn } c \ l) = (A \models \text{conn } c \ l')$ 
proof (cases c rule: connective-cases-arity-2)
  case nullary
  thus  $(A \models \text{conn } c \ l) \longleftrightarrow (A \models \text{conn } c \ l')$  using wf wf' by auto
next
  case unary note c = this
  then obtain a where l:  $l = [a]$  using wf-conn-Not-decomp wf by metis
  obtain a' where l':  $l' = [a']$  using wf-conn-Not-decomp wf' c by metis
  have  $A \models a \longleftrightarrow A \models a'$  using l l' by (metis nth-Cons-0 same)
  thus  $A \models \text{conn } c \ l \longleftrightarrow A \models \text{conn } c \ l'$  using l l' c by auto
next
  case binary note c = this
  then obtain a b where l:  $l = [a, b]$ 
  using wf-conn-bin-list-length list-length2-decomp wf by metis
  obtain a' b' where l':  $l' = [a', b']$ 
  using wf-conn-bin-list-length list-length2-decomp wf' c by metis

  have p:  $A \models a \longleftrightarrow A \models a' \wedge A \models b \longleftrightarrow A \models b'$ 
  using l l' same by (metis diff-Suc-1 nth-Cons' nat.distinct(2))+
  show  $A \models \text{conn } c \ l \longleftrightarrow A \models \text{conn } c \ l'$ 
  using wf c p unfolding binary-connectives-def l l' by auto
qed

Relation between propo-rew-step and the rewriting we have seen before:  $\text{propo-rew-step } r \ \varphi \ \varphi'$ 
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  $\exists \psi \ \psi' p. r \ \psi \ \psi' \wedge \text{path-to } p \ \varphi \ \psi \wedge \text{replace-at } p \ \varphi \ \psi' = \varphi'$ 
  using assms
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi \ \psi$ )
  moreover have path-to []  $\varphi \ \varphi$  by auto
  moreover have replace-at []  $\varphi \ \psi = \psi$  by auto
  ultimately show ?case by metis
next
  case (propo-rew-one-step-lift  $\varphi \ \varphi' c \ \xi \ \xi'$ ) note rel = this(1) and IH0 = this(2) and corr = this(3)
  obtain  $\psi \ \psi' p$  where IH:  $r \ \psi \ \psi' \wedge \text{path-to } p \ \varphi \ \psi \wedge \text{replace-at } p \ \varphi \ \psi' = \varphi'$  using IH0 by metis

  {
    fix x :: 'v
    assume  $c = CT \vee c = CF \vee c = CVar \ x$ 
    hence False using corr by auto
    hence  $\exists \psi \ \psi' p. r \ \psi \ \psi' \wedge \text{path-to } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \ \psi$ 
       $\wedge \text{replace-at } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \ \psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
    by fast
  }
  moreover {
    assume c:  $c = CNot$ 
    hence empty:  $\xi = [] \ \xi' = []$  using corr by auto
  }

```

```

have path-to (L#p) (conn c (ξ@ (φ # ξ'))) ψ
  using c empty IH wf-conn-unary path-to-l by fastforce
moreover have replace-at (L#p) (conn c (ξ@ (φ # ξ'))) ψ' = conn c (ξ@ (φ' # ξ'))
  using c empty IH by auto
ultimately have ∃ ψ ψ' p. r ψ ψ' ∧ path-to p (conn c (ξ@ (φ # ξ'))) ψ
  ∧ replace-at p (conn c (ξ@ (φ # ξ'))) ψ' = conn c (ξ@ (φ' # ξ'))
  using IH by metis
}
moreover {
  assume c: c ∈ binary-connectives
  have length (ξ@ φ # ξ') = 2 using wf-conn-bin-list-length corr c by metis
  hence length ξ + length ξ' = 1 by auto
  hence ld: (length ξ = 1 ∧ length ξ' = 0) ∨ (length ξ = 0 ∧ length ξ' = 1) by arith
  obtain a b where ab: (ξ=[] ∧ ξ'=[b]) ∨ (ξ=[a] ∧ ξ'=[] )
  using ld by (case-tac ξ, case-tac ξ', auto)
  {
    assume φ: ξ=[] ∧ ξ'=[b]
    have path-to (L#p) (conn c (ξ@ (φ # ξ'))) ψ
      using φ c IH ab corr by (simp add: path-to-l)
    moreover have replace-at (L#p) (conn c (ξ@ (φ # ξ'))) ψ' = conn c (ξ@ (φ' # ξ'))
      using c IH ab φ unfolding binary-connectives-def by auto
    ultimately have ∃ ψ ψ' p. r ψ ψ' ∧ path-to p (conn c (ξ@ (φ # ξ'))) ψ
      ∧ replace-at p (conn c (ξ@ (φ # ξ'))) ψ' = conn c (ξ@ (φ' # ξ'))
      using IH by metis
  }
  moreover {
    assume φ: ξ=[a] ξ'=[]
    hence path-to (R#p) (conn c (ξ@ (φ # ξ'))) ψ
      using c IH corr path-to-r corr φ by (simp add: path-to-r)
    moreover have replace-at (R#p) (conn c (ξ@ (φ # ξ'))) ψ' = conn c (ξ@ (φ' # ξ'))
      using c IH ab φ unfolding binary-connectives-def by auto
    ultimately have ?case using IH by metis
  }
  ultimately have ?case using ab by blast
}
ultimately show ?case using connective-cases-arity by blast
qed

```

## 6.2 Consistency preservation

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

**definition** *preserves-un-sat* **where**

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

**lemma** *propo-rew-step-preservers-val-explicit*:

*propo-rew-step*  $r \varphi \psi \implies \text{preserves-un-sat } r \implies \text{propo-rew-step } r \varphi \psi \implies (\forall A. A \models \varphi \longleftrightarrow A \models \psi)$

**unfolding** *preserves-un-sat-def*

**proof** (*induction rule: propo-rew-step.induct*)

**case** *global-rel*

**thus** ?case **by** *simp*

**next**

**case** (*propo-rew-one-step-lift*  $\varphi \varphi' c \xi \xi'$ ) **note**  $\text{rel} = \text{this}(1)$  **and**  $\text{wf} = \text{this}(2)$

**and**  $\text{IH} = \text{this}(3)[\text{OF } \text{this}(4) \text{ this}(1)]$  **and**  $\text{consistent} = \text{this}(4)$

{

```

fix A
from IH have  $\forall n. (A \models (\xi @ \varphi \# \xi') ! n) = (A \models (\xi @ \varphi' \# \xi') ! n)$ 
  by (metis (mono-tags, hide-lams) list-update-length nth-Cons-0 nth-append-length-plus
    nth-list-update-neg)
hence  $(A \models \text{conn } c (\xi @ \varphi \# \xi')) = (A \models \text{conn } c (\xi @ \varphi' \# \xi'))$ 
  by (meson consistency-decompose-into-list wf wf-conn-no-arity-change-helper
    wf-conn-no-arity-change)
}
thus  $\forall A. A \models \text{conn } c (\xi @ \varphi \# \xi') \longleftrightarrow A \models \text{conn } c (\xi @ \varphi' \# \xi')$  by auto
qed

```

```

lemma propo-rew-step-preservers-val':
  assumes preserves-un-sat r
  shows preserves-un-sat (propo-rew-step r)
  using assms by (simp add: preserves-un-sat-def propo-rew-step-preservers-val-explicit)

```

```

lemma preserves-un-sat-OO[intro]:
  preserves-un-sat f  $\implies$  preserves-un-sat g  $\implies$  preserves-un-sat (f OO g)
  unfolding preserves-un-sat-def by auto

```

```

lemma star-consistency-preservation-explicit:
  assumes (propo-rew-step r)**  $\varphi \psi$  and preserves-un-sat r
  shows  $\forall A. A \models \varphi \longleftrightarrow A \models \psi$ 
  using assms by (induct rule: rtranclp-induct)
  (auto simp add: propo-rew-step-preservers-val-explicit)

```

```

lemma star-consistency-preservation:
  preserves-un-sat r  $\implies$  preserves-un-sat (propo-rew-step r)**
  by (simp add: star-consistency-preservation-explicit preserves-un-sat-def)

```

### 6.3 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  $\implies$  preserves-un-sat (full (propo-rew-step r))
  by (metis full-def preserves-un-sat-def star-consistency-preservation)

```

```

lemma full-propo-rew-step-subformula:
  full (propo-rew-step r)  $\varphi' \varphi \implies \neg(\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi')$ 
  unfolding full-def using propo-rew-step-subformula-rec by metis

```

## 7 Transformation testing

### 7.1 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**  $all\_subformula\_st :: ('a \text{ propo} \Rightarrow \text{bool}) \Rightarrow 'a \text{ propo} \Rightarrow \text{bool}$  **where**  
 $all\_subformula\_st \text{ test-symb } \varphi \equiv \forall \psi. \psi \preceq \varphi \longrightarrow \text{test-symb } \psi$

**lemma**  $test\_symb\_imp\_all\_subformula\_st[simp]$ :  
 $test\_symb \text{ FT} \Longrightarrow all\_subformula\_st \text{ test-symb FT}$   
 $test\_symb \text{ FF} \Longrightarrow all\_subformula\_st \text{ test-symb FF}$   
 $test\_symb (FVar \ x) \Longrightarrow all\_subformula\_st \text{ test-symb } (FVar \ x)$   
**unfolding**  $all\_subformula\_st\_def$  **using**  $subformula\_leaf$  **by**  $metis+$

**lemma**  $all\_subformula\_st\_test\_symb\_true\_phi$ :  
 $all\_subformula\_st \text{ test-symb } \varphi \Longrightarrow \text{test-symb } \varphi$   
**unfolding**  $all\_subformula\_st\_def$  **by**  $auto$

**lemma**  $all\_subformula\_st\_decomp\_imp$ :  
 $wf\_conn \ c \ l \Longrightarrow (test\_symb (conn \ c \ l) \wedge (\forall \varphi \in \text{set } l. all\_subformula\_st \text{ test-symb } \varphi))$   
 $\Longrightarrow all\_subformula\_st \text{ test-symb } (conn \ c \ l)$   
**unfolding**  $all\_subformula\_st\_def$  **by**  $auto$

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

**lemma**  $all\_subformula\_st\_decomp\_rec$ :  
 $all\_subformula\_st \text{ test-symb } (conn \ c \ l) \Longrightarrow wf\_conn \ c \ l$   
 $\Longrightarrow (test\_symb (conn \ c \ l) \wedge (\forall \varphi \in \text{set } l. all\_subformula\_st \text{ test-symb } \varphi))$   
**unfolding**  $all\_subformula\_st\_def$  **by**  $auto$

**lemma**  $all\_subformula\_st\_decomp$ :  
**fixes**  $c :: 'v \text{ connective}$  **and**  $l :: 'v \text{ propo list}$   
**assumes**  $wf\_conn \ c \ l$   
**shows**  $all\_subformula\_st \text{ test-symb } (conn \ c \ l)$   
 $\longleftrightarrow (test\_symb (conn \ c \ l) \wedge (\forall \varphi \in \text{set } l. all\_subformula\_st \text{ test-symb } \varphi))$   
**using**  $assms \ all\_subformula\_st\_decomp\_rec \ all\_subformula\_st\_decomp\_imp$  **by**  $metis$

**lemma**  $helper\_fact: c \in \text{binary-connectives} \longleftrightarrow (c = COr \vee c = CAnd \vee c = CEq \vee c = CImp)$   
**unfolding**  $binary\_connectives\_def$  **by**  $auto$

**lemma**  $all\_subformula\_st\_decomp\_explicit[simp]$ :  
**fixes**  $\varphi \ \psi :: 'v \text{ propo}$   
**shows**  $all\_subformula\_st \text{ test-symb } (FAnd \ \varphi \ \psi)$   
 $\longleftrightarrow (test\_symb (FAnd \ \varphi \ \psi) \wedge all\_subformula\_st \text{ test-symb } \varphi \wedge all\_subformula\_st \text{ test-symb } \psi)$   
**and**  $all\_subformula\_st \text{ test-symb } (FOr \ \varphi \ \psi)$   
 $\longleftrightarrow (test\_symb (FOr \ \varphi \ \psi) \wedge all\_subformula\_st \text{ test-symb } \varphi \wedge all\_subformula\_st \text{ test-symb } \psi)$   
**and**  $all\_subformula\_st \text{ test-symb } (FNot \ \varphi)$   
 $\longleftrightarrow (test\_symb (FNot \ \varphi) \wedge all\_subformula\_st \text{ test-symb } \varphi)$   
**and**  $all\_subformula\_st \text{ test-symb } (FEq \ \varphi \ \psi)$   
 $\longleftrightarrow (test\_symb (FEq \ \varphi \ \psi) \wedge all\_subformula\_st \text{ test-symb } \varphi \wedge all\_subformula\_st \text{ test-symb } \psi)$   
**and**  $all\_subformula\_st \text{ test-symb } (FImp \ \varphi \ \psi)$   
 $\longleftrightarrow (test\_symb (FImp \ \varphi \ \psi) \wedge all\_subformula\_st \text{ test-symb } \varphi \wedge all\_subformula\_st \text{ test-symb } \psi)$

**proof** –

**have**  $all\_subformula\_st \text{ test-symb } (FAnd \ \varphi \ \psi) \longleftrightarrow all\_subformula\_st \text{ test-symb } (conn \ CAnd \ [\varphi, \psi])$   
**by**  $auto$   
**moreover have**  $\dots \longleftrightarrow test\_symb (conn \ CAnd \ [\varphi, \psi]) \wedge (\forall \xi \in \text{set } [\varphi, \psi]. all\_subformula\_st \text{ test-symb } \xi)$   
**using**  $all\_subformula\_st\_decomp \ wf\_conn\_helper\_facts(5)$  **by**  $metis$   
**finally show**  $all\_subformula\_st \text{ test-symb } (FAnd \ \varphi \ \psi)$   
 $\longleftrightarrow (test\_symb (FAnd \ \varphi \ \psi) \wedge all\_subformula\_st \text{ test-symb } \varphi \wedge all\_subformula\_st \text{ test-symb } \psi)$

```

by simp

have all-subformula-st test-symb (FOr  $\varphi$   $\psi$ )  $\longleftrightarrow$  all-subformula-st test-symb (conn COr  $[\varphi, \psi]$ )
  by auto
moreover have ...  $\longleftrightarrow$ 
  (test-symb (conn COr  $[\varphi, \psi]$ )  $\wedge$  ( $\forall \xi \in \text{set } [\varphi, \psi]. \text{all-subformula-st test-symb } \xi$ ))
  using all-subformula-st-decomp wf-conn-helper-facts(6) by metis
finally show all-subformula-st test-symb (FOr  $\varphi$   $\psi$ )
   $\longleftrightarrow$  (test-symb (FOr  $\varphi$   $\psi$ )  $\wedge$  all-subformula-st test-symb  $\varphi$   $\wedge$  all-subformula-st test-symb  $\psi$ )
  by simp

have all-subformula-st test-symb (FEq  $\varphi$   $\psi$ )  $\longleftrightarrow$  all-subformula-st test-symb (conn CEq  $[\varphi, \psi]$ )
  by auto
moreover have ...
   $\longleftrightarrow$  (test-symb (conn CEq  $[\varphi, \psi]$ )  $\wedge$  ( $\forall \xi \in \text{set } [\varphi, \psi]. \text{all-subformula-st test-symb } \xi$ ))
  using all-subformula-st-decomp wf-conn-helper-facts(8) by metis
finally show all-subformula-st test-symb (FEq  $\varphi$   $\psi$ )
   $\longleftrightarrow$  (test-symb (FEq  $\varphi$   $\psi$ )  $\wedge$  all-subformula-st test-symb  $\varphi$   $\wedge$  all-subformula-st test-symb  $\psi$ )
  by simp

have all-subformula-st test-symb (FImp  $\varphi$   $\psi$ )  $\longleftrightarrow$  all-subformula-st test-symb (conn CImp  $[\varphi, \psi]$ )
  by auto
moreover have ...
   $\longleftrightarrow$  (test-symb (conn CImp  $[\varphi, \psi]$ )  $\wedge$  ( $\forall \xi \in \text{set } [\varphi, \psi]. \text{all-subformula-st test-symb } \xi$ ))
  using all-subformula-st-decomp wf-conn-helper-facts(7) by metis
finally show all-subformula-st test-symb (FImp  $\varphi$   $\psi$ )
   $\longleftrightarrow$  (test-symb (FImp  $\varphi$   $\psi$ )  $\wedge$  all-subformula-st test-symb  $\varphi$   $\wedge$  all-subformula-st test-symb  $\psi$ )
  by simp

have all-subformula-st test-symb (FNot  $\varphi$ )  $\longleftrightarrow$  all-subformula-st test-symb (conn CNot  $[\varphi]$ )
  by auto
moreover have ... = (test-symb (conn CNot  $[\varphi]$ )  $\wedge$  ( $\forall \xi \in \text{set } [\varphi]. \text{all-subformula-st test-symb } \xi$ ))
  using all-subformula-st-decomp wf-conn-helper-facts(1) by metis
finally show all-subformula-st test-symb (FNot  $\varphi$ )
   $\longleftrightarrow$  (test-symb (FNot  $\varphi$ )  $\wedge$  all-subformula-st test-symb  $\varphi$ ) by simp
qed

```

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

**lemma** *subformula-all-subformula-st*:

```

 $\psi \preceq \varphi \implies \text{all-subformula-st test-symb } \varphi \implies \text{all-subformula-st test-symb } \psi$ 
by (induct rule: subformula.induct, auto simp add: all-subformula-st-decomp)

```

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 propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool and test-symb:: 'v propo  $\Rightarrow$  bool and x :: 'v
and  $\varphi$  :: 'v 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  $(\exists \psi\ \psi'. \psi \preceq \varphi \wedge r\ \psi\ \psi')$ 
using assms
proof (induct  $\varphi$  rule: propo-induct-arity)

```

```

case (nullary  $\varphi$   $x$ )
thus  $\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi'$ 
  using wf-conn-nullary test-symb-false-nullary by fastforce
next
case (unary  $\varphi$ ) note  $IH = \text{this}(1)[OF \text{this}(2)]$  and  $r = \text{this}(2)$  and  $nst = \text{this}(3)$  and  $subf = \text{this}(4)$ 
from  $r \text{ IH } nst$  have  $H: \neg \text{all-subformula-st test-symb } \varphi \implies \exists \psi. \psi \preceq \varphi \wedge (\exists \psi'. r \psi \psi')$ 
  by (metis subformula-in-subformula-not subformula-refl subformula-trans)
{
  assume  $n: \neg \text{test-symb } (FNot \varphi)$ 
  obtain  $\psi$  where  $r (FNot \varphi) \psi$  using subformula-refl  $r n nst$  by blast
  moreover have  $FNot \varphi \preceq FNot \varphi$  using subformula-refl by auto
  ultimately have  $\exists \psi \psi'. \psi \preceq FNot \varphi \wedge r \psi \psi'$  by metis
}
moreover {
  assume  $n: \text{test-symb } (FNot \varphi)$ 
  hence  $\neg \text{all-subformula-st test-symb } \varphi$ 
    using all-subformula-st-decomp-explicit(3)  $nst$   $subf$  by blast
  hence  $\exists \psi \psi'. \psi \preceq FNot \varphi \wedge r \psi \psi'$ 
    using  $H$  subformula-in-subformula-not subformula-refl subformula-trans by blast
}
ultimately show  $\exists \psi \psi'. \psi \preceq FNot \varphi \wedge r \psi \psi'$  by blast
next
case (binary  $\varphi \varphi1 \varphi2$ )
note  $IH\varphi1-0 = \text{this}(1)[OF \text{this}(4)]$  and  $IH\varphi2-0 = \text{this}(2)[OF \text{this}(4)]$  and  $r = \text{this}(4)$ 
  and  $\varphi = \text{this}(3)$  and  $le = \text{this}(5)$  and  $nst = \text{this}(6)$ 

obtain  $c :: 'v$  connective where
   $c: (c = CAnd \vee c = COr \vee c = CImp \vee c = CEq) \wedge \text{conn } c [\varphi1, \varphi2] = \varphi$ 
  using  $\varphi$  by fastforce

hence corr: wf-conn  $c [\varphi1, \varphi2]$  using wf-conn.simps unfolding binary-connectives-def by auto
have inc:  $\varphi1 \preceq \varphi \varphi2 \preceq \varphi$  using binary-connectives-def  $c$  subformula-in-binary-conn by blast+
from  $r \text{ IH}\varphi1-0$  have  $IH\varphi1: \neg \text{all-subformula-st test-symb } \varphi1 \implies \exists \psi \psi'. \psi \preceq \varphi1 \wedge r \psi \psi'$ 
  using inc(1) subformula-trans  $le$  by blast
from  $r \text{ IH}\varphi2-0$  have  $IH\varphi2: \neg \text{all-subformula-st test-symb } \varphi2 \implies \exists \psi. \psi \preceq \varphi2 \wedge (\exists \psi'. r \psi \psi')$ 
  using inc(2) subformula-trans  $le$  by blast
have cases:  $\neg \text{test-symb } \varphi \vee \neg \text{all-subformula-st test-symb } \varphi1 \vee \neg \text{all-subformula-st test-symb } \varphi2$ 
  using  $c nst$  by auto
show  $\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi'$ 
  using  $IH\varphi1 \text{ IH}\varphi2$  subformula-trans inc subformula-refl cases  $le$  by blast
qed

```

## 7.2 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:  $\forall c \xi \varphi \xi' \varphi'. \varphi \preceq \Phi \longrightarrow$



$propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \longrightarrow wf\text{-}conn\ c\ (\xi @ \varphi \# \xi') \longrightarrow test\text{-}symb\ (conn\ c\ (\xi @ \varphi \# \xi')) \longrightarrow$   
 $test\text{-}symb\ \varphi' \longrightarrow test\text{-}symb\ (conn\ c\ (\xi @ \varphi' \# \xi'))$

### 7.2.1 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**  $r:: 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$  **and**  $test\text{-}symb:: 'v\ propo \Rightarrow bool$  **and**  $x:: 'v$   
**and**  $\varphi\ \psi\ \Phi:: 'v\ propo$   
**assumes**  $H: \forall \varphi' \psi. \varphi' \preceq \Phi \longrightarrow r\ \varphi' \psi \longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi'$   
 $\longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \psi$   
**and**  $H': \forall (c:: 'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'. \varphi \preceq \Phi \longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi'$   
 $\longrightarrow wf\text{-}conn\ c\ (\xi @ \varphi \# \xi') \longrightarrow test\text{-}symb\ (conn\ c\ (\xi @ \varphi \# \xi')) \longrightarrow test\text{-}symb\ \varphi'$   
 $\longrightarrow test\text{-}symb\ (conn\ c\ (\xi @ \varphi' \# \xi'))$  **and**  
 $propo\text{-}rew\text{-}step\ r\ \varphi\ \psi$  **and**  
 $\varphi \preceq \Phi$  **and**  
 $all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi$   
**shows**  $all\text{-}subformula\text{-}st\ test\text{-}symb\ \psi$   
**using** *assms(3-5)*  
**proof** (*induct rule: propo-rew-step.induct*)  
**case** *global-rel*  
**thus** *?case using H by simp*  
**next**  
**case** (*propo-rew-one-step-lift*  $\varphi\ \varphi'\ c\ \xi\ \xi'$ )  
**note**  $rel = this(1)$  **and**  $\varphi = this(2)$  **and**  $corr = this(3)$  **and**  $\Phi = this(4)$  **and**  $nst = this(5)$   
**have**  $sq: \varphi \preceq \Phi$   
**using**  $\Phi\ corr\ subformula\text{-}into\text{-}subformula\ subformula\text{-}refl\ subformula\text{-}trans$   
**by** (*metis in-set-conv-decomp*)  
**from**  $corr$  **have**  $\forall \psi. \psi \in set\ (\xi @ \varphi \# \xi') \longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \psi$   
**using**  $all\text{-}subformula\text{-}st\ decomp\ nst$  **by** *blast*  
**hence**  $*$ :  $\forall \psi. \psi \in set\ (\xi @ \varphi' \# \xi') \longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \psi$  **using**  $\varphi\ sq$  **by** *fastforce*  
**hence**  $test\text{-}symb\ \varphi'$  **using**  $all\text{-}subformula\text{-}st\ test\text{-}symb\ true\text{-}\phi$  **by** *auto*  
**moreover from**  $corr\ nst$  **have**  $test\text{-}symb\ (conn\ c\ (\xi @ \varphi \# \xi'))$   
**using**  $all\text{-}subformula\text{-}st\ decomp$  **by** *blast*  
**ultimately have**  $test\text{-}symb: test\text{-}symb\ (conn\ c\ (\xi @ \varphi' \# \xi'))$  **using**  $H'\ sq\ corr\ rel$  **by** *blast*  
  
**have**  $wf\text{-}conn\ c\ (\xi @ \varphi' \# \xi')$   
**by** (*metis wf-conn-no-arity-change-helper corr wf-conn-no-arity-change*)  
**thus**  $all\text{-}subformula\text{-}st\ test\text{-}symb\ (conn\ c\ (\xi @ \varphi' \# \xi'))$   
**using**  $*\ test\text{-}symb$  **by** (*metis all-subformula-st-decomp*)  
**qed**

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

**lemma** *propo-rew-step-inv-stay*:

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

$all\_subformula\_st\ test\_symb\ \varphi$   
**shows**  $all\_subformula\_st\ test\_symb\ \psi$   
**using**  $propo\_rew\_step\_inv\_stay'$  [of  $\varphi\ r\ test\_symb\ \varphi\ \psi$ ] *assms subformula-refl by metis*

The lemmas can be lifted to *full* ( $propo\_rew\_step\ r$ ) instead of  $propo\_rew\_step$

## 7.2.2 Invariant after all rewriting

**lemma** *full-propo-rew-step-inv-stay-with-inc*:

**fixes**  $r:: 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$  **and**  $test\_symb:: 'v\ propo \Rightarrow bool$  **and**  $x:: 'v$   
**and**  $\varphi\ \psi:: 'v\ propo$

**assumes**

$H: \forall\ \varphi\ \psi. propo\_rew\_step\ r\ \varphi\ \psi \longrightarrow all\_subformula\_st\ test\_symb\ \varphi$   
 $\longrightarrow all\_subformula\_st\ test\_symb\ \psi$  **and**

$H': \forall\ (c:: 'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'. \varphi \preceq \Phi \longrightarrow propo\_rew\_step\ r\ \varphi\ \varphi'$   
 $\longrightarrow wf\_conn\ c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow test\_symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow test\_symb\ \varphi'$   
 $\longrightarrow test\_symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$  **and**  
 $\varphi \preceq \Phi$  **and**

**full**:  $full\ (propo\_rew\_step\ r)\ \varphi\ \psi$  **and**

**init**:  $all\_subformula\_st\ test\_symb\ \varphi$

**shows**  $all\_subformula\_st\ test\_symb\ \psi$

**using** *assms unfolding full-def*

**proof** –

**have**  $rel: (propo\_rew\_step\ r)^{**}\ \varphi\ \psi$   
**using** *full unfolding full-def by auto*

**thus**  $all\_subformula\_st\ test\_symb\ \psi$

**using** *init*

**proof** (*induct rule: rtrancplp-induct*)

**case** *base*

**then show**  $all\_subformula\_st\ test\_symb\ \varphi$  **by** *blast*

**next**

**case** (*step b c*) **note**  $star = this(1)$  **and**  $IH = this(3)$  **and**  $one = this(2)$  **and**  $all = this(4)$

**then have**  $all\_subformula\_st\ test\_symb\ b$  **by** *metis*

**then show**  $all\_subformula\_st\ test\_symb\ c$  **using**  $propo\_rew\_step\_inv\_stay'\ H\ H'\ rel\ one$  **by** *auto*

**qed**

**qed**

**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**

$H: \forall\ \varphi\ \psi. propo\_rew\_step\ r\ \varphi\ \psi \longrightarrow all\_subformula\_st\ test\_symb\ \varphi$   
 $\longrightarrow all\_subformula\_st\ test\_symb\ \psi$  **and**

$H': \forall\ (c:: 'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'. propo\_rew\_step\ r\ \varphi\ \varphi' \longrightarrow wf\_conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')$   
 $\longrightarrow test\_symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow test\_symb\ \varphi' \longrightarrow test\_symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$  **and**

**full**:  $full\ (propo\_rew\_step\ r)\ \varphi\ \psi$  **and**

**init**:  $all\_subformula\_st\ test\_symb\ \varphi$

**shows**  $all\_subformula\_st\ test\_symb\ \psi$

**using** *full-propo-rew-step-inv-stay-with-inc* [of  $r\ test\_symb\ \varphi$ ] *assms subformula-refl by metis*

**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**

$H: \forall\ \varphi\ \psi. r\ \varphi\ \psi \longrightarrow all\_subformula\_st\ test\_symb\ \varphi \longrightarrow all\_subformula\_st\ test\_symb\ \psi$  **and**

$H': \forall\ (c:: 'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'. wf\_conn\ c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow test\_symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi'))$

```

    → test-symb  $\varphi'$  → test-symb (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  $\psi$ 
unfolding full-def
proof -
  have rel: (propo-rew-step r)**  $\varphi \psi$ 
    using full unfolding full-def by auto
  thus all-subformula-st test-symb  $\psi$ 
    using init
  proof (induct rule: rtrancp-induct)
    case base
      thus all-subformula-st test-symb  $\varphi$  by blast
    next
      case (step b c)
        note star = this(1) and IH = this(3) and one = this(2) and all = this(4)
        hence all-subformula-st test-symb b by metis
        thus all-subformula-st test-symb c
          using propo-rew-step-inv-stay subformula-refl H H' rel one by auto
        qed
      qed
  qed

lemma full-propo-rew-step-inv-stay-conn:
  fixes r:: 'v propo ⇒ 'v propo ⇒ bool and test-symb:: 'v propo ⇒ bool and x:: 'v
  and  $\varphi \psi$ :: 'v propo
  assumes
    H:  $\forall \varphi \psi. r \varphi \psi \rightarrow$  all-subformula-st test-symb  $\varphi \rightarrow$  all-subformula-st test-symb  $\psi$  and
    H':  $\forall (c:: 'v \text{ connective}) l l'. \text{wf-conn } c \ l \rightarrow \text{wf-conn } c \ l' \rightarrow$ 
      ( $\text{test-symb (conn } c \ l) \longleftrightarrow \text{test-symb (conn } c \ l')$ ) 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 -
  have  $\bigwedge (c:: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{wf-conn } c \ (\xi @ \varphi \# \xi') \Rightarrow$ 
    test-symb (conn c ( $\xi @ \varphi \# \xi'$ ))  $\Rightarrow$  test-symb  $\varphi' \Rightarrow$  test-symb (conn c ( $\xi @ \varphi' \# \xi'$ ))
    using H' by (metis wf-conn-no-arity-change-helper wf-conn-no-arity-change)
  thus all-subformula-st test-symb  $\psi$ 
    using H full init full-propo-rew-step-inv-stay by blast
  qed

end
theory Prop-Normalisation
imports Main Prop-Logic Prop-Abstract-Transformation
begin

```

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

## 8 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.

## 8.1 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*:  
 $A \models \text{FEq } \varphi \ \psi \longleftrightarrow A \models \text{FAnd } (\text{FImp } \varphi \ \psi) \ (\text{FImp } \psi \ \varphi)$   
**by** *auto*

**lemma** *elim-equiv-explicit*: *elim-equiv*  $\varphi \ \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$   
**by** (*induct rule: elim-equiv.induct, auto*)

**lemma** *elim-equiv-consistent*: *preserves-un-sat elim-equiv*  
**unfolding** *preserves-un-sat-def* **by** (*simp add: elim-equiv-explicit*)

**lemma** *elimEquiv-lifted-consistant*:  
*preserves-un-sat (full (propo-rew-step elim-equiv))*  
**by** (*simp add: elim-equiv-consistent*)

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

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

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

**lemma** *no-equiv-symb-conn-characterization*[simp]:  
**fixes** *c* :: 'v connective **and** *l* :: 'v propo list  
**assumes** *wf*: *wf-conn c l*  
**shows** *no-equiv-symb (conn c l)  $\longleftrightarrow c \neq \text{CEq}$*   
**by** (*metis connective.distinct(13,25,35,43) wf no-equiv-symb.elims(3) no-equiv-symb.simps(1) wf-conn.cases wf-conn-list(6)*)

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

**lemma** *no-equiv-eq*[simp]:  
**fixes**  $\varphi \ \psi$  :: 'v propo  
**shows**  
 $\neg \text{no-equiv } (\text{FEq } \varphi \ \psi)$   
 $\text{no-equiv } \text{FT}$   
 $\text{no-equiv } \text{FF}$   
**using** *no-equiv-symb.simps(1) all-subformula-st-test-symb-true-phi* **unfolding** *no-equiv-def* **by** *auto*

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 propo  
**shows**  
 $\text{no-equiv } (\text{FNot } \varphi) \longleftrightarrow \text{no-equiv } \varphi$   
 $\text{no-equiv } (\text{FAnd } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$   
 $\text{no-equiv } (\text{FOr } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$   
 $\text{no-equiv } (\text{FImp } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$   
**by** (*auto simp add: no-equiv-def*)

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  $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elim-equiv } \psi \psi'$ 
proof –
  have test-symb-false-nullary:
     $\forall x::'v. \text{no-equiv-symb } FF \wedge \text{no-equiv-symb } FT \wedge \text{no-equiv-symb } (FVar\ x)$ 
  unfolding no-equiv-def by auto
  moreover {
    fix  $c::'v \text{ connective}$  and  $l::'v \text{ propo list}$  and  $\psi::'v \text{ propo}$ 
    assume  $a1: \text{elim-equiv } (\text{conn } c\ l)\ \psi$ 
    have  $\bigwedge p\ pa. \neg \text{elim-equiv } (p::'v \text{ propo})\ pa \vee \neg \text{no-equiv-symb } p$ 
    using elim-equiv.cases no-equiv-symb.simps(1) by blast
    hence  $\text{elim-equiv } (\text{conn } c\ l)\ \psi \implies \neg \text{no-equiv-symb } (\text{conn } c\ l)$  using  $a1$  by metis
  }
  moreover have  $H': \forall \psi. \neg \text{elim-equiv } FT\ \psi \vee \forall \psi. \neg \text{elim-equiv } FF\ \psi \vee \forall \psi\ x. \neg \text{elim-equiv } (FVar\ x)\ \psi$ 
  using elim-equiv.cases by auto
  moreover have  $\bigwedge \varphi. \neg \text{no-equiv-symb } \varphi \implies \exists \psi. \text{elim-equiv } \varphi\ \psi$ 
  by (case-tac  $\varphi$ , auto simp add: elim-equiv.simps)
  hence  $\bigwedge \varphi'. \varphi' \preceq \varphi \implies \neg \text{no-equiv-symb } \varphi' \implies \exists \psi. \text{elim-equiv } \varphi'\ \psi$  by force
  ultimately show ?thesis
  using no-test-symb-step-exists no-equiv test-symb-false-nullary unfolding no-equiv-def by blast
qed

```

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

```

lemma no-equiv-full-propo-rew-step-elim-equiv:
  full (propo-rew-step elim-equiv)  $\varphi\ \psi \implies \text{no-equiv } \psi$ 
  using full-propo-rew-step-subformula no-equiv-elim-equiv-step by blast

```

## 8.2 Eliminate Implication

After that, we can eliminate the implication symbols.

```

inductive elim-imp ::  $'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  where
  [simp]: elim-imp (FImp  $\varphi\ \psi$ ) (FOr (FNot  $\varphi$ )  $\psi$ )

```

```

lemma elim-imp-transformation-consistent:
   $A \models FImp\ \varphi\ \psi \longleftrightarrow A \models FOr\ (FNot\ \varphi)\ \psi$ 
by auto

```

```

lemma elim-imp-explicit: elim-imp  $\varphi\ \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$ 
by (induct  $\varphi\ \psi$  rule: elim-imp.induct, auto)

```

```

lemma elim-imp-consistent: preserves-un-sat elim-imp
  unfolding preserves-un-sat-def by (simp add: elim-imp-explicit)

```

```

lemma elim-imp-lifted-consistant:
  preserves-un-sat (full (propo-rew-step elim-imp))
by (simp add: elim-imp-consistent)

```

```

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

```

```

lemma no-imp-symb-conn-characterization:
  wf-conn c l  $\implies$  no-imp-symb (conn c l)  $\longleftrightarrow$  c  $\neq$  CImp
  by (induction rule: wf-conn-induct) auto

```

```

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

```

```

lemma no-imp-Imp[simp]:
   $\neg$ no-imp (FImp  $\varphi$   $\psi$ )
  no-imp FT
  no-imp FF
  unfolding no-imp-def by auto

```

```

lemma all-subformula-st-decomp-explicit-imp[simp]:
fixes  $\varphi$   $\psi :: 'v$  propo
shows
  no-imp (FNot  $\varphi$ )  $\longleftrightarrow$  no-imp  $\varphi$ 
  no-imp (FAnd  $\varphi$   $\psi$ )  $\longleftrightarrow$  (no-imp  $\varphi \wedge$  no-imp  $\psi$ )
  no-imp (FOr  $\varphi$   $\psi$ )  $\longleftrightarrow$  (no-imp  $\varphi \wedge$  no-imp  $\psi$ )
  by auto

```

Invariant of the *elim-imp* transformation

```

lemma elim-imp-no-equiv:
  elim-imp  $\varphi$   $\psi \implies$  no-equiv  $\varphi \implies$  no-equiv  $\psi$ 
  by (induct  $\varphi$   $\psi$  rule: elim-imp.induct, auto)

```

```

lemma elim-imp-inv:
  fixes  $\varphi$   $\psi :: 'v$  propo
  assumes full (propo-rew-step elim-imp)  $\varphi$   $\psi$ 
  and no-equiv  $\varphi$ 
  shows no-equiv  $\psi$ 
  using full-propo-rew-step-inv-stay-conn[of elim-imp no-equiv-symb  $\varphi$   $\psi$ ] assms elim-imp-no-equiv
  no-equiv-symb-conn-characterization unfolding no-equiv-def by metis

```

```

lemma no-no-imp-elim-imp-step-exists:

```

```

  fixes  $\varphi :: 'v$  propo
  assumes no-equiv:  $\neg$  no-imp  $\varphi$ 
  shows  $\exists \psi \psi'. \psi \preceq \varphi \wedge$  elim-imp  $\psi \psi'$ 

```

**proof** –

```

  have test-symb-false-nullary:  $\forall x. \text{no-imp-symb } FF \wedge \text{no-imp-symb } FT \wedge \text{no-imp-symb } (FVar (x:: 'v))$ 
  by auto

```

```

  moreover {
    fix c:: 'v connective and l:: 'v propo list and  $\psi :: 'v$  propo
    have H: elim-imp (conn c l)  $\psi \implies \neg$ no-imp-symb (conn c l)
    by (auto elim: elim-imp.cases)
  }

```

**moreover**

```

  have H':  $\forall \psi. \neg$ elim-imp FT  $\psi \forall \psi. \neg$ elim-imp FF  $\psi \forall \psi x. \neg$ elim-imp (FVar x)  $\psi$ 
  by (auto elim: elim-imp.cases)+

```

```

moreover have  $\bigwedge \varphi. \neg$  no-imp-symb  $\varphi \implies \exists \psi. \text{elim-imp } \varphi \psi$ 

```

```

    apply (case-tac  $\varphi$ ) using elim-imp.simps by force+
  hence ( $\bigwedge \varphi'. \varphi' \preceq \varphi \implies \neg \text{no-imp-symb } \varphi' \implies \exists \psi. \text{elim-imp } \varphi' \psi$ ) by force
  ultimately show ?thesis
    using no-test-symb-step-exists no-equiv test-symb-false-nullary unfolding no-imp-def by blast
qed

```

```

lemma no-imp-full-propo-rew-step-elim-imp: full (propo-rew-step elim-imp)  $\varphi \psi \implies \text{no-imp } \psi$ 
  using full-propo-rew-step-subformula no-no-imp-elim-imp-step-exists by blast

```

### 8.3 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)  $\varphi$  |
ElimTB1': elimTB (FAnd FT  $\varphi$ )  $\varphi$  |

```

```

ElimTB2: elimTB (FAnd  $\varphi$  FF) FF |
ElimTB2': elimTB (FAnd FF  $\varphi$ ) FF |

```

```

ElimTB3: elimTB (FOr  $\varphi$  FT) FT |
ElimTB3': elimTB (FOr FT  $\varphi$ ) FT |

```

```

ElimTB4: elimTB (FOr  $\varphi$  FF)  $\varphi$  |
ElimTB4': elimTB (FOr FF  $\varphi$ )  $\varphi$  |

```

```

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

```

**lemma** *elimTB-consistent: preserves-un-sat elimTB*

**proof** –

```

{
  fix  $\varphi \psi :: 'b \text{ propo}$ 
  have elimTB  $\varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$  by (induct-tac rule: elimTB.inducts) auto
}
thus ?thesis using preserves-un-sat-def by auto
qed

```

**inductive** *no-T-F-symb* :: *'v propo  $\Rightarrow$  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  $c \ \psi s \implies \text{no-T-F-symb } (\text{conn } c \ \psi s) \longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in \text{set } \psi s. \psi \neq FF \wedge \psi \neq FT))$ 

```

```

unfolding no-T-F-symb.simps apply (cases c)
  using wf-conn-list(1) apply fastforce
  using wf-conn-list(2) apply fastforce
  using wf-conn-list(3) apply fastforce
  apply (metis (no-types, hide-lams) conn-inj connective.distinct(5,17))
  using conn-inj apply blast+
done

```

**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*  $\varphi$   $\psi$ )  $\longleftrightarrow (\forall \chi \in \text{set } [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$   
**apply** (*metis* *conn.simps*(36) *conn.simps*(37) *conn.simps*(5) *propo.distinct*(19) *wf-conn-helper-facts*(5) *wf-conn-no-T-F-symb-iff*)  
**apply** (*metis* *conn.simps*(36) *conn.simps*(37) *conn.simps*(6) *propo.distinct*(22) *wf-conn-helper-facts*(6) *wf-conn-no-T-F-symb-iff*)  
**using** *wf-conn-no-T-F-symb-iff* **apply** *fastforce*  
**by** (*metis* *conn.simps*(36) *conn.simps*(37) *conn.simps*(7) *propo.distinct*(23) *wf-conn-helper-facts*(7) *wf-conn-no-T-F-symb-iff*)

**lemma** *no-T-F-symb-false*[simp]:  
**fixes** *c* :: 'v *connective*  
**shows**  
 $\neg \text{no-T-F-symb } (FT :: 'v \text{ propo})$   
 $\neg \text{no-T-F-symb } (FF :: 'v \text{ propo})$   
**by** (*metis* (*no-types*) *conn.simps*(1,2) *wf-conn-no-T-F-symb-iff* *wf-conn-nullary*) +

**lemma** *no-T-F-symb-bool*[simp]:  
**fixes** *x* :: 'v  
**shows** *no-T-F-symb* (*FVar* *x*)  
**using** *no-T-F-symb-comp* *wf-conn-nullary* **by** (*metis* *connective.distinct*(3, 15) *conn.simps*(3) *empty-iff* *list.set*(1))

**lemma** *no-T-F-symb-fnot-imp*:  
 $\neg \text{no-T-F-symb } (FNot \varphi) \implies \varphi = FT \vee \varphi = FF$   
**proof** (*rule ccontr*)  
**assume** *n*:  $\neg \text{no-T-F-symb } (FNot \varphi)$   
**assume**  $\neg (\varphi = FT \vee \varphi = FF)$   
**hence**  $\forall \varphi' \in \text{set } [\varphi]. \varphi' \neq FT \wedge \varphi' \neq FF$  **by** *auto*  
**moreover** **have** *wf-conn* *CNot*  $[\varphi]$  **by** *simp*  
**ultimately** **have** *no-T-F-symb* (*FNot*  $\varphi$ )  
**using** *no-T-F-symb.intros* **by** (*metis* *conn.simps*(4) *connective.distinct*(5,17))  
**thus** *False* **using** *n* **by** *blast*  
**qed**

**lemma** *no-T-F-symb-fnot*[simp]:  
 $\text{no-T-F-symb } (FNot \varphi) \longleftrightarrow \neg (\varphi = FT \vee \varphi = FF)$   
**using** *no-T-F-symb.simps* *no-T-F-symb-fnot-imp* **by** (*metis* *conn-inj-not*(2) *list.set-intros*(1))

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*[simp]:  
**fixes** *x* :: 'v  
**shows** *no-T-F-symb-except-toplevel* (*FVar* *x*)



by *simp*

**lemma** *no-T-F-symb-except-toplevel-not-decom*:

$\varphi \neq FT \implies \varphi \neq FF \implies \text{no-T-F-symb-except-toplevel } (F\text{Not } \varphi)$

by *simp*

**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*  $c$   $[\varphi, \psi]$ )

**by** (*metis* (*no-types*, *lifting*) *assms*  $c$  *conn.simps*(4) *list.discI* *noTrue-no-T-F-symb-except-toplevel* *wf-conn-no-T-F-symb-iff* *no-T-F-symb-fnot* *set.ConsD* *wf-conn-binary* *wf-conn-helper-facts*(1) *wf-conn-list-decomp*(1,2))

**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**  $\neg \text{no-T-F-symb-except-toplevel } (\text{conn } c \ l)$

**by** (*metis* *assms* *empty-iff* *no-T-F-symb-except-toplevel.simps* *wf-conn-no-T-F-symb-iff* *set-empty* *wf-conn-list*(1,2))

**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**

$\neg \text{no-T-F-symb-except-toplevel } (F\text{And } \varphi \ \psi)$

$\neg \text{no-T-F-symb-except-toplevel } (F\text{Or } \varphi \ \psi)$

$\neg \text{no-T-F-symb-except-toplevel } (F\text{Imp } \varphi \ \psi)$

$\neg \text{no-T-F-symb-except-toplevel } (F\text{Eq } \varphi \ \psi)$

**using** *assms* *no-T-F-symb-except-toplevel-if-is-a-true-false* **unfolding** *binary-connectives-def*

**by** (*metis* (*no-types*) *conn.simps*(5–8) *insert-iff* *list.simps*(14–15) *wf-conn-helper-facts*(5–8))+

**lemma** *no-T-F-symb-except-top-level-false-not*[*simp*]:

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

**assumes**  $\varphi = FT \vee \varphi = FF$

**shows**

$\neg \text{no-T-F-symb-except-toplevel } (F\text{Not } \varphi)$

**by** (*simp* *add*: *assms* *no-T-F-symb-except-toplevel.simps*)

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

**definition** *no-T-F-except-top-level* **where**

*no-T-F-except-top-level*  $\equiv \text{all-subformula-st } \text{no-T-F-symb-except-toplevel}$

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

**definition** *no-T-F* **where**

*no-T-F*  $\equiv \text{all-subformula-st } \text{no-T-F-symb}$

**lemma** *no-T-F-except-top-level-false*:

**fixes**  $l :: 'v \text{ propo list}$  **and**  $c :: 'v \text{ connective}$   
**assumes**  $wf\text{-conn } c \ l$   
**and**  $FT \in \text{set } l \vee FF \in \text{set } l$   
**shows**  $\neg no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ (conn \ c \ l)$   
**by** (*simp add: all-subformula-st-decomp assms no-T-F-except-top-level-def*  
 $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\text{-}if\text{-}is\text{-}a\text{-}true\text{-}false$ )

**lemma**  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\text{-}false\text{-}example[simp]$ :  
**fixes**  $\varphi \ \psi :: 'v \text{ propo}$   
**assumes**  $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$   
**shows**  
 $\neg no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ (FAnd \ \varphi \ \psi)$   
 $\neg no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ (FOr \ \varphi \ \psi)$   
 $\neg no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ (FEq \ \varphi \ \psi)$   
 $\neg no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ (FImp \ \varphi \ \psi)$   
**by** (*metis all-subformula-st-test-symb-true-phi assms no-T-F-except-top-level-def*  
 $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}top\text{-}level\text{-}false\text{-}example$ ) $+$

**lemma**  $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\text{-}no\text{-}T\text{-}F\text{-}symb$ :  
 $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel \ \varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies no\text{-}T\text{-}F\text{-}symb \ \varphi$   
**by** (*induct rule: no-T-F-symb-except-toplevel.induct, auto*)

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

**lemma**  $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\text{-}all\text{-}subformula\text{-}st\text{-}no\text{-}T\text{-}F\text{-}symb$ :  
 $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ \varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies no\text{-}T\text{-}F \ \varphi$   
**unfolding**  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\text{-}def$   $no\text{-}T\text{-}F\text{-}def$  **apply** (*induct*  $\varphi$ )  
**using**  $no\text{-}T\text{-}F\text{-}symb\text{-}fnot$  **by** *fastforce* $+$

**lemma**  $no\text{-}T\text{-}F\text{-}no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level$ :  
 $no\text{-}T\text{-}F \ \varphi \implies no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ \varphi$   
**unfolding**  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\text{-}def$   $no\text{-}T\text{-}F\text{-}def$   
**unfolding**  $all\text{-}subformula\text{-}st\text{-}def$  **by** *auto*

**lemma**  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\text{-}simp[simp]$ :  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ FF \ no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ FT$   
**unfolding**  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\text{-}def$  **by** *auto*

**lemma**  $no\text{-}T\text{-}F\text{-}no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level'[simp]$ :  
 $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level \ \varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee no\text{-}T\text{-}F \ \varphi)$   
**apply** *auto*  
**using**  $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\text{-}all\text{-}subformula\text{-}st\text{-}no\text{-}T\text{-}F\text{-}symb$   $no\text{-}T\text{-}F\text{-}no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level$   
**by** *blast* $+$

**lemma**  $no\text{-}T\text{-}F\text{-}bin\text{-}decomp[simp]$ :  
**assumes**  $c: c \in \text{binary-connectives}$   
**shows**  $no\text{-}T\text{-}F \ (conn \ c \ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F \ \varphi \wedge no\text{-}T\text{-}F \ \psi)$   
**proof** –  
**have**  $wf: wf\text{-conn } c \ [\varphi, \psi]$  **using**  $c$  **by** *auto*  
**hence**  $no\text{-}T\text{-}F \ (conn \ c \ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F\text{-}symb \ (conn \ c \ [\varphi, \psi]) \wedge no\text{-}T\text{-}F \ \varphi \wedge no\text{-}T\text{-}F \ \psi)$   
**by** (*simp add: all-subformula-st-decomp no-T-F-def*)  
**thus**  $no\text{-}T\text{-}F \ (conn \ c \ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F \ \varphi \wedge no\text{-}T\text{-}F \ \psi)$   
**using**  $c \ wf \ all\text{-}subformula\text{-}st\text{-}decomp \ list.discI \ no\text{-}T\text{-}F\text{-}def \ no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\text{-}bin\text{-}decom$   
 $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\text{-}no\text{-}T\text{-}F\text{-}symb \ no\text{-}T\text{-}F\text{-}symb\text{-}false(1,2) \ wf\text{-conn-helper-facts}(2,3)$

*wf-conn-list(1,2)* **by** *metis*  
**qed**

**lemma** *no-T-F-bin-decomp-expanded[simp]*:  
**assumes** *c*:  $c = CAnd \vee c = COr \vee c = CEq \vee c = CImp$   
**shows** *no-T-F* (*conn* *c* [ $\varphi$ ,  $\psi$ ])  $\longleftrightarrow$  (*no-T-F*  $\varphi \wedge$  *no-T-F*  $\psi$ )  
**using** *no-T-F-bin-decomp* *assms* **unfolding** *binary-connectives-def* **by** *blast*

**lemma** *no-T-F-comp-expanded-explicit[simp]*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**shows**  
*no-T-F* (*FAnd*  $\varphi \psi$ )  $\longleftrightarrow$  (*no-T-F*  $\varphi \wedge$  *no-T-F*  $\psi$ )  
*no-T-F* (*FOr*  $\varphi \psi$ )  $\longleftrightarrow$  (*no-T-F*  $\varphi \wedge$  *no-T-F*  $\psi$ )  
*no-T-F* (*FEq*  $\varphi \psi$ )  $\longleftrightarrow$  (*no-T-F*  $\varphi \wedge$  *no-T-F*  $\psi$ )  
*no-T-F* (*FImp*  $\varphi \psi$ )  $\longleftrightarrow$  (*no-T-F*  $\varphi \wedge$  *no-T-F*  $\psi$ )  
**using** *assms* *conn.simps(5-8)* *no-T-F-bin-decomp-expanded* **by** (*metis* (*no-types*))+

**lemma** *no-T-F-comp-not[simp]*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**shows** *no-T-F* (*FNot*  $\varphi$ )  $\longleftrightarrow$  *no-T-F*  $\varphi$   
**by** (*metis* *all-subformula-st-decomp-explicit(3)* *all-subformula-st-test-symb-true-phi* *no-T-F-def* *no-T-F-symb-false(1,2)* *no-T-F-symb-fnot-imp*)

**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*  $\varphi$   
**using** *assms* **by** *auto*

**lemma** *no-T-F-decomp-not*:  
**fixes**  $\varphi :: 'v \text{ propo}$   
**assumes**  $\varphi$ : *no-T-F* (*FNot*  $\varphi$ )  
**shows** *no-T-F*  $\varphi$   
**using** *assms* **by** *auto*

**lemma** *no-T-F-symb-except-toplevel-step-exists*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes** *no-equiv*  $\varphi$  **and** *no-imp*  $\varphi$   
**shows**  $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTB } \psi \psi'$   
**proof** (*induct*  $\psi$  *rule*: *propo-induct-arity*)  
**case** (*nullary*  $\varphi' x$ )  
**hence** *False* **using** *no-T-F-symb-except-toplevel-true* *no-T-F-symb-except-toplevel-false* **by** *auto*  
**thus** ?*case* **by** *blast*  
**next**  
**case** (*unary*  $\psi$ )  
**hence**  $\psi = FF \vee \psi = FT$  **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*  
**thus** ?*case* **using** *ElimTB5* *ElimTB6* **by** *blast*  
**next**  
**case** (*binary*  $\varphi' \psi_1 \psi_2$ )  
**note** *IH1* = *this(1)* **and** *IH2* = *this(2)* **and**  $\varphi' = \text{this}(3)$  **and**  $F\varphi = \text{this}(4)$  **and**  $n = \text{this}(5)$   
**{**  
**assume**  $\varphi' = FImp \psi_1 \psi_2 \vee \varphi' = FEq \psi_1 \psi_2$   
**hence** *False* **using**  $n \ F\varphi$  *subformula-all-subformula-st* *assms* **by** (*metis* (*no-types*) *no-equiv-eq(1)* *no-equiv-def* *no-imp-imp(1)* *no-imp-def*)  
**}**

```

  hence ?case by blast
}
moreover {
  assume  $\varphi'$ :  $\varphi' = FAnd \ \psi1 \ \psi2 \vee \varphi' = FOr \ \psi1 \ \psi2$ 
  hence  $\psi1 = FT \vee \psi2 = FT \vee \psi1 = FF \vee \psi2 = FF$ 
  using no-T-F-symb-except-toplevel-bin-decom conn.simps(5,6) n unfolding binary-connectives-def
  by fastforce+
  hence ?case using elimTB.intros  $\varphi'$  by blast
}
ultimately show ?case using  $\varphi'$  by blast
qed

```

**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  $\exists \psi \ \psi'. \ \psi \preceq \varphi \wedge \text{elimTB } \psi \ \psi'$ 
proof -
  have test-symb-false-nullary:  $\forall x. \text{no-T-F-symb-except-toplevel } (FF :: 'v \text{ propo})$ 
     $\wedge \text{no-T-F-symb-except-toplevel } FT \wedge \text{no-T-F-symb-except-toplevel } (FVar \ (x :: 'v))$  by auto
  moreover {
    fix c:: 'v connective and l:: 'v propo list and  $\psi :: 'v \text{ propo}$ 
    have H:  $\text{elimTB } (\text{conn } c \ l) \ \psi \implies \neg \text{no-T-F-symb-except-toplevel } (\text{conn } c \ l)$ 
      by (case-tac (conn c l) rule: elimTB.cases, auto)
  }
  moreover {
    fix x:: 'v
    have H':  $\text{no-T-F-except-top-level } FT \ \text{no-T-F-except-top-level } FF$ 
       $\text{no-T-F-except-top-level } (FVar \ x)$ 
      by (auto simp add: no-T-F-except-top-level-def test-symb-false-nullary)
  }
  moreover {
    fix  $\psi$ 
    have  $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTB } \psi \ \psi'$ 
      using no-T-F-symb-except-toplevel-step-exists no-equiv no-imp by auto
  }
  ultimately show ?thesis
    using no-test-symb-step-exists noTB unfolding no-T-F-except-top-level-def by blast
qed

```

**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  $\psi$ 
proof -
  {
    fix  $\varphi \ \psi :: 'v \text{ propo}$ 
    have H:  $\text{elimTB } \varphi \ \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$ 
      by (induct  $\varphi \ \psi$  rule: elimTB.induct, auto)
  }
  thus no-equiv  $\psi$ 
    using full-propo-rew-step-inv-stay-conn[of elimTB no-equiv-symb  $\varphi \ \psi$ ]
      no-equiv-symb-conn-characterization assms unfolding no-equiv-def by metis
next

```

```

{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have  $H: \text{elimTB } \varphi \psi \implies \text{no-imp } \varphi \implies \text{no-imp } \psi$ 
    by (induct  $\varphi \psi$  rule:  $\text{elimTB.induct}$ , auto)
}
thus  $\text{no-imp } \psi$ 
  using  $\text{full-propo-rew-step-inv-stay-conn[of elimTB no-imp-symb } \varphi \psi] \text{ assms}$ 
     $\text{no-imp-symb-conn-characterization unfolding no-imp-def by metis}$ 
qed

```

**lemma** *elimTB-full-propo-rew-step*:  
**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes**  $\text{no-equiv } \varphi$  **and**  $\text{no-imp } \varphi$  **and**  $\text{full (propo-rew-step elimTB) } \varphi \psi$   
**shows**  $\text{no-T-F-except-top-level } \psi$   
**using**  $\text{full-propo-rew-step-subformula no-T-F-except-top-level-rew assms elimTB-inv by fastforce}$

## 8.4 PushNeg

Push the negation inside the formula, until the litteral.

**inductive** *pushNeg* **where**

```

PushNeg1[simp]:  $\text{pushNeg (FNot (FAnd } \varphi \psi)) (FOr (FNot \varphi) (FNot \psi)) \mid$ 
PushNeg2[simp]:  $\text{pushNeg (FNot (FOr } \varphi \psi)) (FAnd (FNot \varphi) (FNot \psi)) \mid$ 
PushNeg3[simp]:  $\text{pushNeg (FNot (FNot } \varphi)) \varphi$ 

```

**lemma** *pushNeg-transformation-consistent*:

```

 $A \models \text{FNot (FAnd } \varphi \psi) \longleftrightarrow A \models (\text{FOr (FNot } \varphi) (\text{FNot } \psi))$ 
 $A \models \text{FNot (FOr } \varphi \psi) \longleftrightarrow A \models (\text{FAnd (FNot } \varphi) (\text{FNot } \psi))$ 
 $A \models \text{FNot (FNot } \varphi) \longleftrightarrow A \models \varphi$ 
by auto

```

**lemma** *pushNeg-explicit*:  $\text{pushNeg } \varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$   
**by** (induct  $\varphi \psi$  rule:  $\text{pushNeg.induct}$ , auto)

**lemma** *pushNeg-consistent*:  $\text{preserves-un-sat pushNeg}$   
**unfolding**  $\text{preserves-un-sat-def by (simp add: pushNeg-explicit)}$

**lemma** *pushNeg-lifted-consistant*:

```

 $\text{preserves-un-sat (full (propo-rew-step pushNeg))}$ 
by (simp add: pushNeg-consistent)

```

**fun** *simple* **where**

```

simple FT = True |
simple FF = True |
simple (FVar _) = True |
simple - = False

```

**lemma** *simple-decomp*:

```

 $\text{simple } \varphi \longleftrightarrow (\varphi = \text{FT} \vee \varphi = \text{FF} \vee (\exists x. \varphi = \text{FVar } x))$ 
by (case-tac } \varphi, \text{ auto)}

```

**lemma** *subformula-conn-decomp-simple*:

```

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

```

```

assumes  $s$ : simple  $\psi$ 
shows  $\varphi \preceq \text{FNot } \psi \longleftrightarrow (\varphi = \text{FNot } \psi \vee \varphi = \psi)$ 
proof –
  have  $\varphi \preceq \text{conn CNot } [\psi] \longleftrightarrow (\varphi = \text{conn CNot } [\psi] \vee (\exists \psi \in \text{set } [\psi]. \varphi \preceq \psi))$ 
    using subformula-conn-decomp wf-conn-helper-facts(1) by metis
  thus  $\varphi \preceq \text{FNot } \psi \longleftrightarrow (\varphi = \text{FNot } \psi \vee \varphi = \psi)$  using  $s$  by (auto simp add: simple-decomp)
qed

```

```

lemma subformula-conn-decomp-explicit[simp]:
  fixes  $\varphi :: 'v \text{ propo}$  and  $x :: 'v$ 
  shows
     $\varphi \preceq \text{FNot } FT \longleftrightarrow (\varphi = \text{FNot } FT \vee \varphi = FT)$ 
     $\varphi \preceq \text{FNot } FF \longleftrightarrow (\varphi = \text{FNot } FF \vee \varphi = FF)$ 
     $\varphi \preceq \text{FNot } (\text{FVar } x) \longleftrightarrow (\varphi = \text{FNot } (\text{FVar } x) \vee \varphi = \text{FVar } x)$ 
  by (auto simp add: subformula-conn-decomp-simple)

```

```

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

```

```

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

```

```

lemma simple-not-Not[simp]:
   $\neg \text{simple-not } (\text{FNot } (\text{FAnd } \varphi \psi))$ 
   $\neg \text{simple-not } (\text{FNot } (\text{FOr } \varphi \psi))$ 
by auto

```

```

lemma simple-not-step-exists:
  fixes  $\varphi \psi :: 'v \text{ propo}$ 
  assumes no-equiv  $\varphi$  and no-imp  $\varphi$ 
  shows  $\psi \preceq \varphi \implies \neg \text{simple-not-symb } \psi \implies \exists \psi'. \text{pushNeg } \psi \psi'$ 
  apply (induct  $\psi$ , auto)
  apply (case-tac  $\psi$ , auto intro: pushNeg.intros)
  by (metis assms(1,2) no-imp-Imp(1) no-equiv-eq(1) no-imp-def no-equiv-def
    subformula-in-subformula-not subformula-all-subformula-st)+

```

```

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  $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{pushNeg } \psi \psi'$ 
proof –
  have  $\forall x. \text{simple-not-symb } (FF :: 'v \text{ propo}) \wedge \text{simple-not-symb } FT \wedge \text{simple-not-symb } (\text{FVar } (x :: 'v))$ 
    by auto
  moreover {
    fix  $c :: 'v \text{ connective}$  and  $l :: 'v \text{ propo list}$  and  $\psi :: 'v \text{ propo}$ 
    have  $H: \text{pushNeg } (\text{conn } c \ l) \ \psi \implies \neg \text{simple-not-symb } (\text{conn } c \ l)$ 
      by (case-tac (conn  $c \ l$ ) rule: pushNeg.cases, simp-all)
  }
  moreover {
    fix  $x :: 'v$ 
    have  $H': \text{simple-not } FT \text{ simple-not } FF \text{ simple-not } (\text{FVar } x)$ 
      by simp-all
  }

```

```

}
moreover {
  fix  $\psi :: 'v \text{ propo}$ 
  have  $\psi \preceq \varphi \implies \neg \text{simple-not-symb } \psi \implies \exists \psi'. \text{pushNeg } \psi \psi'$ 
  using simple-not-step-exists no-equiv no-imp by blast
}
ultimately show ?thesis using no-test-symb-step-exists noTB unfolding simple-not-def by blast
qed

```

**lemma** *no-T-F-except-top-level-pushNeg1*:

```

 $\text{no-T-F-except-top-level } (F\text{Not } (F\text{And } \varphi \psi)) \implies \text{no-T-F-except-top-level } (F\text{Or } (F\text{Not } \varphi) (F\text{Not } \psi))$ 
using no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-comp-not no-T-F-decomp(1)
 $\text{no-T-F-decomp(2) no-T-F-no-T-F-except-top-level}$  by (metis no-T-F-comp-expanded-explicit(2)
 $\text{propo.distinct(5,17)}$ )

```

**lemma** *no-T-F-except-top-level-pushNeg2*:

```

 $\text{no-T-F-except-top-level } (F\text{Not } (F\text{Or } \varphi \psi)) \implies \text{no-T-F-except-top-level } (F\text{And } (F\text{Not } \varphi) (F\text{Not } \psi))$ 
by auto

```

**lemma** *no-T-F-symb-pushNeg*:

```

 $\text{no-T-F-symb } (F\text{Or } (F\text{Not } \varphi') (F\text{Not } \psi'))$ 
 $\text{no-T-F-symb } (F\text{And } (F\text{Not } \varphi') (F\text{Not } \psi'))$ 
 $\text{no-T-F-symb } (F\text{Not } (F\text{Not } \varphi'))$ 
by auto

```

**lemma** *propo-rew-step-pushNeg-no-T-F-symb*:

```

 $\text{propo-rew-step pushNeg } \varphi \psi \implies \text{no-T-F-except-top-level } \varphi \implies \text{no-T-F-symb } \varphi \implies \text{no-T-F-symb } \psi$ 
apply (induct rule: propo-rew-step.induct)
apply (cases rule: pushNeg.cases)
apply simp-all
apply (metis no-T-F-symb-pushNeg(1))
apply (metis no-T-F-symb-pushNeg(2))
apply (simp, metis all-subformula-st-test-symb-true-phi no-T-F-def)

```

**proof** –

```

fix  $\varphi \varphi':: 'a \text{ propo}$  and  $c:: 'a \text{ connective}$  and  $\xi \xi':: 'a \text{ propo list}$ 
assume rel: propo-rew-step pushNeg  $\varphi \varphi'$ 
and IH:  $\text{no-T-F } \varphi \implies \text{no-T-F-symb } \varphi \implies \text{no-T-F-symb } \varphi'$ 
and wf:  $\text{wf-conn } c (\xi @ \varphi \# \xi')$ 
and  $n: \text{conn } c (\xi @ \varphi \# \xi') = FF \vee \text{conn } c (\xi @ \varphi \# \xi') = FT \vee \text{no-T-F } (\text{conn } c (\xi @ \varphi \# \xi'))$ 
and  $x: c \neq CF \wedge c \neq CT \wedge \varphi \neq FF \wedge \varphi \neq FT \wedge (\forall \psi \in \text{set } \xi \cup \text{set } \xi'. \psi \neq FF \wedge \psi \neq FT)$ 
hence  $c \neq CF \wedge c \neq CT \wedge \text{wf-conn } c (\xi @ \varphi' \# \xi')$ 
  using wf-conn-no-arity-change-helper wf-conn-no-arity-change by metis
moreover have  $n': \text{no-T-F } (\text{conn } c (\xi @ \varphi \# \xi'))$  using n by (simp add: wf wf-conn-list(1,2))
moreover
{
  have  $\text{no-T-F } \varphi$ 
  by (metis Un-iff all-subformula-st-decomp list.set-intros(1) n' wf no-T-F-def set-append)
  moreover hence  $\text{no-T-F-symb } \varphi$ 
  by (simp add: all-subformula-st-test-symb-true-phi no-T-F-def)
  ultimately have  $\varphi' \neq FF \wedge \varphi' \neq FT$ 
  using IH no-T-F-symb-false(1) no-T-F-symb-false(2) by blast
  hence  $\forall \psi \in \text{set } (\xi @ \varphi' \# \xi'). \psi \neq FF \wedge \psi \neq FT$  using x by auto
}
ultimately show  $\text{no-T-F-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$  by (simp add: x)
qed

```

**lemma** *propo-rew-step-pushNeg-no-T-F*:  
*propo-rew-step pushNeg  $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$*   
**proof** (*induct rule: propo-rew-step.induct*)  
 case *global-rel*  
 thus ?case  
 by (*metis* (*no-types, lifting*) *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*  
*no-T-F-def no-T-F-except-top-level-pushNeg1 no-T-F-except-top-level-pushNeg2*  
*no-T-F-no-T-F-except-top-level all-subformula-st-decomp-explicit(3) pushNeg.simps*  
*simple.simps(1,2,5,6)*)  
**next**  
 case (*propo-rew-one-step-lift  $\varphi \varphi' c \xi \xi'$* )  
 note *rel = this(1)* and *IH = this(2)* and *wf = this(3)* and *no-T-F = this(4)*  
 moreover have *wf'*: *wf-conn c ( $\xi @ \varphi' \# \xi'$ )*  
 using *wf-conn-no-arity-change wf-conn-no-arity-change-helper wf* by *metis*  
 ultimately show *no-T-F (conn c ( $\xi @ \varphi' \# \xi'$ ))* unfolding *no-T-F-def*  
 apply(*simp add: all-subformula-st-decomp wf wf'*)  
 using *all-subformula-st-test-symb-true-phi no-T-F-symb-false(1) no-T-F-symb-false(2)* by *blast*  
 qed

**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  $\psi$*   
**proof** –  
 {  
 fix  $\varphi \psi :: 'v \text{ propo}$   
 assume *rel: propo-rew-step pushNeg  $\varphi \psi$*   
 and *no: no-T-F-except-top-level  $\varphi$*   
 hence *no-T-F-except-top-level  $\psi$*   
 proof –  
 {  
 assume  $\varphi = FT \vee \varphi = FF$   
 from *rel this* have *False*  
 apply (*induct rule: propo-rew-step.induct*)  
 using *pushNeg.cases* apply *blast*  
 using *wf-conn-list(1) wf-conn-list(2)* by *auto*  
 hence *no-T-F-except-top-level  $\psi$*  by *blast*  
 }  
 moreover {  
 assume  $\varphi \neq FT \wedge \varphi \neq FF$   
 hence *no-T-F  $\varphi$*  by (*metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*)  
 hence *no-T-F  $\psi$*  using *propo-rew-step-pushNeg-no-T-F rel* by *auto*  
 hence *no-T-F-except-top-level  $\psi$*  by (*simp add: no-T-F-no-T-F-except-top-level*)  
 }  
 ultimately show *no-T-F-except-top-level  $\psi$*  by *metis*  
 qed  
 }  
 moreover {  
 fix  $c :: 'v \text{ connective}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $\zeta \zeta' :: 'v \text{ propo}$   
 assume *rel: propo-rew-step pushNeg  $\zeta \zeta'$*   
 and *incl:  $\zeta \preceq \varphi$*   
 and *corr: wf-conn c ( $\xi @ \zeta \# \xi'$ )*



```

and no-T-F: no-T-F-symb-except-toplevel (conn c (ξ @ ζ # ξ'))
and n: no-T-F-symb-except-toplevel ζ'
have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
proof
  have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
    using corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F
    by blast
  have l: ∀ φ ∈ set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
    using corr wf-conn-no-T-F-symb-iff p by blast
  from rel incl have ζ' ≠ FT ∧ ζ' ≠ FF
    apply (induction ζ ζ' rule: propo-rew-step.induct)
    apply (cases rule: pushNeg.cases, auto)
    by (metis assms(4) no-T-F-symb-except-top-level-false-not no-T-F-except-top-level-def
      all-subformula-st-test-symb-true-phi subformula-in-subformula-not
      subformula-all-subformula-st append-is-Nil-conv list.distinct(1)
      wf-conn-no-arity-change-helper wf-conn-list(1,2) wf-conn-no-arity-change)+
  hence ∀ φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
  moreover have c ≠ CT ∧ c ≠ CF using corr by auto
  ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
    by (metis corr no-T-F-symb-comp wf-conn-no-arity-change wf-conn-no-arity-change-helper)
  qed
}
ultimately show no-T-F-except-top-level ψ
  using full-propo-rew-step-inv-stay-with-inc[of pushNeg no-T-F-symb-except-toplevel φ] assms
  subformula-refl unfolding no-T-F-except-top-level-def full-unfold by metis
next
{
  fix φ ψ :: 'v propo
  have H: pushNeg φ ψ ⇒ no-equiv φ ⇒ no-equiv ψ
    by (induct φ ψ rule: pushNeg.induct, auto)
}
thus no-equiv ψ
  using full-propo-rew-step-inv-stay-conn[of pushNeg no-equiv-symb φ ψ]
  no-equiv-symb-conn-characterization assms unfolding no-equiv-def full-unfold by metis
next
{
  fix φ ψ :: 'v propo
  have H: pushNeg φ ψ ⇒ no-imp φ ⇒ no-imp ψ
    by (induct φ ψ rule: pushNeg.induct, auto)
}
thus no-imp ψ
  using full-propo-rew-step-inv-stay-conn[of pushNeg no-imp-symb φ ψ] assms
  no-imp-symb-conn-characterization unfolding no-imp-def full-unfold by metis
qed

```

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

```

fixes φ ψ :: 'v propo
assumes
  no-equiv φ and
  no-imp φ and
  full (propo-rew-step pushNeg) φ ψ and
  no-T-F-except-top-level φ
shows simple-not ψ
using assms full-propo-rew-step-subformula pushNeg-inv(1,2) simple-not-rew by blast

```

## 8.5 Push inside

**inductive** *push-conn-inside* :: 'v connective  $\Rightarrow$  'v connective  $\Rightarrow$  'v propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool

**for** *c c'* :: 'v connective **where**

*push-conn-inside-l[simp]*:  $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$

$\Longrightarrow$  *push-conn-inside* *c c'* (*conn* *c* [*conn* *c'* [ $\varphi 1$ ,  $\varphi 2$ ],  $\psi$ ])  
 (*conn* *c'* [*conn* *c* [ $\varphi 1$ ,  $\psi$ ], *conn* *c* [ $\varphi 2$ ,  $\psi$ ]]) |

*push-conn-inside-r[simp]*:  $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$

$\Longrightarrow$  *push-conn-inside* *c c'* (*conn* *c* [ $\psi$ , *conn* *c'* [ $\varphi 1$ ,  $\varphi 2$ ]])  
 (*conn* *c'* [*conn* *c* [ $\psi$ ,  $\varphi 1$ ], *conn* *c* [ $\psi$ ,  $\varphi 2$ ]])

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

**by** (*induct*  $\varphi \psi$  *rule*: *push-conn-inside.induct*, *auto*)

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

**unfolding** *preserves-un-sat-def* **by** (*simp* *add*: *push-conn-inside-explicit*)

**lemma** *propo-rew-step-push-conn-inside[simp]*:

$\neg$ *propo-rew-step* (*push-conn-inside* *c c'*) *FT*  $\psi \neg$ *propo-rew-step* (*push-conn-inside* *c c'*) *FF*  $\psi$

**proof** –

```
{
  {
    fix  $\varphi \psi$ 
    have push-conn-inside c c'  $\varphi \psi \Longrightarrow \varphi = FT \vee \varphi = FF \Longrightarrow False$ 
      by (induct rule: push-conn-inside.induct, auto)
  } note H = this
  fix  $\varphi$ 
  have propo-rew-step (push-conn-inside c c')  $\varphi \psi \Longrightarrow \varphi = FT \vee \varphi = FF \Longrightarrow False$ 
    apply (induct rule: propo-rew-step.induct, auto simp add: wf-conn-list(1) wf-conn-list(2))
    using H by blast+
```

**thus**

$\neg$ *propo-rew-step* (*push-conn-inside* *c c'*) *FT*  $\psi$   
 $\neg$ *propo-rew-step* (*push-conn-inside* *c c'*) *FF*  $\psi$  **by** *blast*+

**qed**

**inductive** *not-c-in-c'-symb* :: 'v connective  $\Rightarrow$  'v connective  $\Rightarrow$  'v propo  $\Rightarrow$  bool **for** *c c'* **where**

*not-c-in-c'-symb-l[simp]*: *wf-conn* *c* [*conn* *c'* [ $\varphi$ ,  $\varphi'$ ],  $\psi$ ]  $\Longrightarrow$  *wf-conn* *c'* [ $\varphi$ ,  $\varphi'$ ]

$\Longrightarrow$  *not-c-in-c'-symb* *c c'* (*conn* *c* [*conn* *c'* [ $\varphi$ ,  $\varphi'$ ],  $\psi$ ]) |

*not-c-in-c'-symb-r[simp]*: *wf-conn* *c* [ $\psi$ , *conn* *c'* [ $\varphi$ ,  $\varphi'$ ]]  $\Longrightarrow$  *wf-conn* *c'* [ $\varphi$ ,  $\varphi'$ ]

$\Longrightarrow$  *not-c-in-c'-symb* *c c'* (*conn* *c* [ $\psi$ , *conn* *c'* [ $\varphi$ ,  $\varphi'$ ]])

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

**lemma** *c-in-c'-symb-simp*:

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

**apply** (*induct* *rule*: *not-c-in-c'-symb.induct*, *auto* *simp* *add*: *wf-conn.simps* *wf-conn-list*(1–3))

**using** *conn-inj-not*(2) *wf-conn-binary* **unfolding** *binary-connectives-def* **by** *fastforce*+

**lemma** *c-in-c'-symb-simp'[simp]*:

$\neg$ *not-c-in-c'-symb* *c c'* *FF*

$\neg$ *not-c-in-c'-symb* *c c'* *FT*

$\neg \text{not-c-in-c'-symb } c \ c' \ (FVar \ x)$   
 $\neg \text{not-c-in-c'-symb } c \ c' \ (FNot \ FF)$   
 $\neg \text{not-c-in-c'-symb } c \ c' \ (FNot \ FT)$   
 $\neg \text{not-c-in-c'-symb } c \ c' \ (FNot \ (FVar \ x))$   
**using**  $c\text{-in-c'-symb-simp}$  **by**  $\text{metis+}$

**definition**  $c\text{-in-c'-only}$  **where**

$c\text{-in-c'-only } c \ c' \equiv \text{all-subformula-st } (c\text{-in-c'-symb } c \ c')$

**lemma**  $c\text{-in-c'-only-simp}[simp]:$

$c\text{-in-c'-only } c \ c' \ FF$   
 $c\text{-in-c'-only } c \ c' \ FT$   
 $c\text{-in-c'-only } c \ c' \ (FVar \ x)$   
 $c\text{-in-c'-only } c \ c' \ (FNot \ FF)$   
 $c\text{-in-c'-only } c \ c' \ (FNot \ FT)$   
 $c\text{-in-c'-only } c \ c' \ (FNot \ (FVar \ x))$   
**unfolding**  $c\text{-in-c'-only-def}$  **by**  $\text{auto}$

**lemma**  $\text{not-c-in-c'-symb-commute}:$

$\text{not-c-in-c'-symb } c \ c' \ \xi \implies \text{wf-conn } c \ [\varphi, \psi] \implies \xi = \text{conn } c \ [\varphi, \psi]$   
 $\implies \text{not-c-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$

**proof** (*induct rule: not-c-in-c'-symb.induct*)

**case** ( $\text{not-c-in-c'-symb-r } \varphi' \ \varphi'' \ \psi'$ ) **note**  $H = \text{this}$   
**hence**  $\psi: \psi = \text{conn } c' \ [\varphi'', \psi']$  **using**  $\text{conn-inj}$  **by**  $\text{auto}$   
**have**  $\text{wf-conn } c \ [\text{conn } c' \ [\varphi'', \psi'], \varphi]$   
**using**  $H(1)$   $\text{wf-conn-no-arity-change length-Cons}$  **by**  $\text{metis}$   
**thus**  $\text{not-c-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$   
**unfolding**  $\psi$  **using**  $\text{not-c-in-c'-symb.intros}(1)$   $H$  **by**  $\text{auto}$

**next**

**case** ( $\text{not-c-in-c'-symb-l } \varphi' \ \varphi'' \ \psi'$ ) **note**  $H = \text{this}$   
**hence**  $\varphi = \text{conn } c' \ [\varphi', \varphi'']$  **using**  $\text{conn-inj}$  **by**  $\text{auto}$   
**moreover have**  $\text{wf-conn } c \ [\psi', \text{conn } c' \ [\varphi', \varphi'']]$   
**using**  $H(1)$   $\text{wf-conn-no-arity-change length-Cons}$  **by**  $\text{metis}$   
**ultimately show**  $\text{not-c-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$   
**using**  $\text{not-c-in-c'-symb.intros}(2)$   $\text{conn-inj}$   $\text{not-c-in-c'-symb-l.hyps}$   
 $\text{not-c-in-c'-symb-l.prem}(1,2)$  **by**  $\text{blast}$

**qed**

**lemma**  $\text{not-c-in-c'-symb-commute}':$

$\text{wf-conn } c \ [\varphi, \psi] \implies c\text{-in-c'-symb } c \ c' \ (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow c\text{-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$   
**using**  $\text{not-c-in-c'-symb-commute}$   $\text{wf-conn-no-arity-change}$  **by** ( $\text{metis length-Cons}$ )

**lemma**  $\text{not-c-in-c'-comm}:$

**assumes**  $\text{wf}: \text{wf-conn } c \ [\varphi, \psi]$   
**shows**  $c\text{-in-c'-only } c \ c' \ (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow c\text{-in-c'-only } c \ c' \ (\text{conn } c \ [\psi, \varphi])$  (**is**  $?A \longleftrightarrow ?B$ )

**proof** –

**have**  $?A \longleftrightarrow (c\text{-in-c'-symb } c \ c' \ (\text{conn } c \ [\varphi, \psi])$   
 $\quad \wedge (\forall \xi \in \text{set } [\varphi, \psi]. \text{all-subformula-st } (c\text{-in-c'-symb } c \ c' \ \xi))$   
**using**  $\text{all-subformula-st-decomp wf}$  **unfolding**  $c\text{-in-c'-only-def}$  **by**  $\text{fastforce}$   
**also have**  $\dots \longleftrightarrow (c\text{-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$   
 $\quad \wedge (\forall \xi \in \text{set } [\psi, \varphi]. \text{all-subformula-st } (c\text{-in-c'-symb } c \ c' \ \xi))$   
**using**  $\text{not-c-in-c'-symb-commute}' \text{ wf}$  **by**  $\text{auto}$   
**also**  
**have**  $\text{wf-conn } c \ [\psi, \varphi]$  **using**  $\text{wf-conn-no-arity-change wf}$  **by** ( $\text{metis length-Cons}$ )

hence  $(c\text{-in-}c'\text{-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$   
 $\wedge (\forall \xi \in \text{set } [\psi, \varphi]. \text{all-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$   
 $\longleftrightarrow ?B$   
 using *all-subformula-st-decomp* **unfolding** *c-in-c'-only-def* **by** *fastforce*  
 finally show *?thesis* .  
 qed

**lemma** *not-c-in-c'-simp[simp]*:  
**fixes**  $\varphi 1 \ \varphi 2 \ \psi :: 'v \text{ propo}$  **and**  $x :: 'v$   
**shows**  
 $c\text{-in-}c'\text{-symb } c \ c' \ FT$   
 $c\text{-in-}c'\text{-symb } c \ c' \ FF$   
 $c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$   
 $wf\text{-conn } c \ [\text{conn } c' \ [\varphi 1, \varphi 2], \psi] \implies wf\text{-conn } c' \ [\varphi 1, \varphi 2]$   
 $\implies \neg c\text{-in-}c'\text{-only } c \ c' \ (\text{conn } c \ [\text{conn } c' \ [\varphi 1, \varphi 2], \psi])$   
**apply** (*simp-all add: c-in-c'-only-def*)  
**using** *all-subformula-st-test-symb-true-phi not-c-in-c'-symb-l* **by** *blast*

**lemma** *c-in-c'-symb-not[simp]*:  
**fixes**  $c \ c' :: 'v \text{ connective}$  **and**  $\psi :: 'v \text{ propo}$   
**shows**  $c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ \psi)$   
**proof** –  
 $\{$   
**fix**  $\xi :: 'v \text{ propo}$   
**have**  $not\text{-}c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ \psi) \implies False$   
**apply** (*induct FNot  $\psi$  rule: not-c-in-c'-symb.induct*)  
**using** *conn-inj-not(2)* **by** *blast+*  
 $\}$   
**thus** *?thesis* **by** *auto*  
 qed

**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**  $\psi \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi \implies \exists \psi'. \text{push-conn-inside } c \ c' \ \psi \ \psi'$   
**apply** (*induct  $\psi$  rule: propo-induct-arity*)  
**apply** *auto[2]*  
**proof** –  
**fix**  $\psi 1 \ \psi 2 \ \varphi' :: 'v \text{ propo}$   
**assume** *IH $\psi 1$* :  $\psi 1 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi 1 \implies Ex \ (\text{push-conn-inside } c \ c' \ \psi 1)$   
**and** *IH $\psi 2$* :  $\psi 2 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi 2 \implies Ex \ (\text{push-conn-inside } c \ c' \ \psi 2)$   
**and**  $\varphi': \varphi' = FAnd \ \psi 1 \ \psi 2 \vee \varphi' = FOr \ \psi 1 \ \psi 2 \vee \varphi' = FImp \ \psi 1 \ \psi 2 \vee \varphi' = FEq \ \psi 1 \ \psi 2$   
**and** *in $\varphi$* :  $\varphi' \preceq \varphi$  **and** *n0*:  $\neg c\text{-in-}c'\text{-symb } c \ c' \ \varphi'$   
**hence** *n*:  $not\text{-}c\text{-in-}c'\text{-symb } c \ c' \ \varphi'$  **by** *auto*  
 $\{$   
**assume**  $\varphi': \varphi' = \text{conn } c \ [\psi 1, \psi 2]$   
**obtain**  $a \ b$  **where**  $\psi 1 = \text{conn } c' \ [a, b] \vee \psi 2 = \text{conn } c' \ [a, b]$   
**using** *n  $\varphi'$*  **apply** (*induct rule: not-c-in-c'-symb.induct*)  
**using** *c* **by** *force+*  
**hence**  $Ex \ (\text{push-conn-inside } c \ c' \ \varphi')$   
**unfolding**  $\varphi'$  **apply** *auto*  
**using** *push-conn-inside.intros(1) c c'* **apply** *blast*  
**using** *push-conn-inside.intros(2) c c'* **by** *blast*  
 $\}$   
**moreover**  $\{$

```

assume  $\varphi'$ :  $\varphi' \neq \text{conn } c [\psi 1, \psi 2]$ 
have  $\forall \varphi \ c \ ca. \exists \varphi 1 \ \psi 1 \ \psi 2 \ \psi 1' \ \psi 2' \ \varphi 2'. \text{conn } (c::'v \text{ connective}) [\varphi 1, \text{conn } ca [\psi 1, \psi 2]] = \varphi$ 
 $\vee \text{conn } c [\text{conn } ca [\psi 1', \psi 2'], \varphi 2'] = \varphi \vee c\text{-in-}c'\text{-symb } c \ ca \ \varphi$ 
by (metis not-c-in-c'-symb.cases)
hence  $\exists x \ (\text{push-conn-inside } c \ c' \ \varphi')$ 
by (metis (no-types)  $c \ c' \ n \ \text{push-conn-inside-l} \ \text{push-conn-inside-r}$ )
}
ultimately show  $\exists x \ (\text{push-conn-inside } c \ c' \ \varphi')$  by blast
qed

```

**lemma** *c-in-c'-symb-rew*:

```

fixes  $\varphi :: 'v \text{ propo}$ 
assumes noTB:  $\neg c\text{-in-}c'\text{-only } c \ c' \ \varphi$ 
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 –
have test-symb-false-nullary:
 $\forall x. c\text{-in-}c'\text{-symb } c \ c' \ (FF::'v \text{ propo}) \wedge c\text{-in-}c'\text{-symb } c \ c' \ FT$ 
 $\wedge c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ (x::'v))$ 
by auto
moreover {
fix  $x :: 'v$ 
have  $H': c\text{-in-}c'\text{-symb } c \ c' \ FT \ c\text{-in-}c'\text{-symb } c \ c' \ FF \ c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$ 
by simp+
}
moreover {
fix  $\psi :: 'v \text{ propo}$ 
have  $\psi \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi \implies \exists \psi'. \text{push-conn-inside } c \ c' \ \psi \ \psi'$ 
by (auto simp add: assms(2)  $c' \ c\text{-in-}c'\text{-symb-step-exists}$ )
}
ultimately show ?thesis using noTB no-test-symb-step-exists[of  $c\text{-in-}c'\text{-symb } c \ c'$ ]
unfolding  $c\text{-in-}c'\text{-only-def}$  by metis
qed

```

**lemma** *push-conn-insidec-in-c'-symb-no-T-F*:

```

fixes  $\varphi \ \psi :: 'v \text{ propo}$ 
shows propo-rew-step ( $\text{push-conn-inside } c \ c' \ \varphi \ \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$ )
proof (induct rule: propo-rew-step.induct)
case (global-rel  $\varphi \ \psi$ )
thus  $\text{no-T-F } \psi$ 
by (cases rule: push-conn-inside.cases, auto)
next
case (propo-rew-one-step-lift  $\varphi \ \varphi' \ c \ \xi \ \xi'$ )
note  $\text{rel} = \text{this}(1)$  and  $\text{IH} = \text{this}(2)$  and  $\text{wf} = \text{this}(3)$  and  $\text{no-T-F} = \text{this}(4)$ 
have  $\text{no-T-F } \varphi$ 
using  $\text{wf no-T-F no-T-F-def subformula-into-subformula subformula-all-subformula-st}$ 
 $\text{subformula-refl}$  by (metis (no-types) in-set-conv-decomp)
hence  $\varphi': \text{no-T-F } \varphi' \text{ using IH by blast}$ 

have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{no-T-F } \zeta$  by (metis wf no-T-F no-T-F-def all-subformula-st-decomp)
hence  $n: \forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \text{no-T-F } \zeta$  using  $\varphi'$  by auto
hence  $n': \forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \zeta \neq FF \wedge \zeta \neq FT$ 
using  $\varphi'$  by (metis no-T-F-symb-false(1) no-T-F-symb-false(2) no-T-F-def
all-subformula-st-test-symb-true-phi)

```

```

have wf': wf-conn c (ξ @ φ' # ξ')
  using wf wf-conn-no-arity-change by (metis wf-conn-no-arity-change-helper)
{
  fix x :: 'v
  assume c = CT ∨ c = CF ∨ c = CVar x
  hence False using wf by auto
  hence no-T-F (conn c (ξ @ φ' # ξ')) by blast
}
moreover {
  assume c: c = CNot
  hence ξ = [] ξ' = [] using wf by auto
  hence no-T-F (conn c (ξ @ φ' # ξ'))
    using c by (metis φ' conn.simps(4) no-T-F-symb-false(1,2) no-T-F-symb-fnot no-T-F-def
      all-subformula-st-decomp-explicit(3) all-subformula-st-test-symb-true-phi self-append-conv2)
}
moreover {
  assume c: c ∈ binary-connectives
  hence no-T-F-symb (conn c (ξ @ φ' # ξ')) using wf' n' no-T-F-symb.simps by fastforce
  hence no-T-F (conn c (ξ @ φ' # ξ')) by (metis all-subformula-st-decomp-imp wf' n no-T-F-def)
}
ultimately show no-T-F (conn c (ξ @ φ' # ξ')) using connective-cases-arity by auto
qed

```

**lemma** *simple-propo-rew-step-push-conn-inside-inv*:

```

propo-rew-step (push-conn-inside c c') φ ψ ⇒ simple φ ⇒ simple ψ
  apply (induct rule: propo-rew-step.induct)
  apply (case-tac φ, auto simp add: push-conn-inside.simps)[1]
  by (metis append-is-Nil-conv list.distinct(1) simple.elims(2) wf-conn-list(1-3))

```

**lemma** *simple-propo-rew-step-inv-push-conn-inside-simple-not*:

```

fixes c c' :: 'v connective and φ ψ :: 'v propo
shows propo-rew-step (push-conn-inside c c') φ ψ ⇒ simple-not φ ⇒ simple-not ψ
proof (induct rule: propo-rew-step.induct)
  case (global-rel φ ψ)
  thus ?case by (case-tac φ, auto simp add: push-conn-inside.simps)
next
  case (propo-rew-one-step-lift φ φ' ca ξ ξ')
  thus ?case
    proof (case-tac ca rule: connective-cases-arity, auto)
      fix φ φ' :: 'v propo and c :: 'v connective and ξ ξ' :: 'v propo list
      assume rel: propo-rew-step (push-conn-inside c c') φ φ'
      assume simple φ
      thus simple φ' using rel simple-propo-rew-step-push-conn-inside-inv by blast
    next
      fix φ φ' :: 'v propo and ca :: 'v connective and ξ ξ' :: 'v propo list
      assume rel: propo-rew-step (push-conn-inside c c') φ φ'
      and IH: all-subformula-st simple-not-symb φ ⇒ all-subformula-st simple-not-symb φ'
      and wf: wf-conn ca (ξ @ φ # ξ')
      and simple-not: all-subformula-st simple-not-symb (conn ca (ξ @ φ # ξ'))
      and ca: ca ∈ binary-connectives

      obtain a b where ab: ξ @ φ' # ξ' = [a, b]

```

```

    using wf ca list-length2-decomp wf-conn-bin-list-length
    by (metis (no-types) wf-conn-no-arity-change-helper)
have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi')$ . simple-not  $\zeta$ 
    by (metis wf all-subformula-st-decomp simple-not simple-not-def)
hence  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi')$ . simple-not  $\zeta$  by (simp add: IH)
moreover have simple-not-symb (conn ca ( $\xi @ \varphi' \# \xi'$ )) using ca
    by (metis ab conn.simps(5-8) helper-fact simple-not-symb.simps(5) simple-not-symb.simps(6)
        simple-not-symb.simps(7) simple-not-symb.simps(8))
ultimately show all-subformula-st simple-not-symb (conn ca ( $\xi @ \varphi' \# \xi'$ ))
    by (simp add: ab all-subformula-st-decomp ca)
qed
qed

```

**lemma** *propo-rew-step-push-conn-inside-simple-not*:

```

fixes  $\varphi \varphi' :: 'v \text{ propo}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $c :: 'v \text{ connective}$ 
shows propo-rew-step (push-conn-inside c c')  $\varphi \varphi' \implies \text{wf-conn } c (\xi @ \varphi \# \xi')$ 
 $\implies \text{simple-not-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \implies \text{simple-not-symb } \varphi'$ 
 $\implies \text{simple-not-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
apply (induct rule: propo-rew-step.induct)
apply (metis (no-types, lifting) append-eq-append-conv2 append-self-conv conn.simps(4)
    conn-inj-not(1) global-rel simple-not-symb.elims(3) simple-not-symb.simps(1)
    simple-propo-rew-step-push-conn-inside-inv wf-conn-list-decomp(4) wf-conn-no-arity-change
    wf-conn-no-arity-change-helper)

```

**proof** (*case-tac* c *rule: connective-cases-arity*, *auto*)

```

fix  $\varphi \varphi' :: 'v \text{ propo}$  and ca ::  $'v \text{ connective}$  and  $\chi s \chi s' :: 'v \text{ propo list}$ 
assume simple-not-symb (conn c ( $\xi @ \text{conn } ca (\chi s @ \varphi \# \chi s') \# \xi'$ ))
and simple-not-symb (conn ca ( $\chi s @ \varphi' \# \chi s'$ ))
and corr: wf-conn c ( $\xi @ \text{conn } ca (\chi s @ \varphi \# \chi s') \# \xi'$ )
and c:  $c \in \text{binary-connectives}$ 
have corr': wf-conn c ( $\xi @ \text{conn } ca (\chi s @ \varphi' \# \chi s') \# \xi'$ )
    using corr wf-conn-no-arity-change by (metis wf-conn-no-arity-change-helper)
obtain a b where  $\xi @ \text{conn } ca (\chi s @ \varphi' \# \chi s') \# \xi' = [a, b]$ 
    using corr' c list-length2-decomp wf-conn-bin-list-length by metis
thus simple-not-symb (conn c ( $\xi @ \text{conn } ca (\chi s @ \varphi' \# \chi s') \# \xi'$ ))
    using c unfolding binary-connectives-def by auto

```

**next**

```

fix  $\varphi \varphi' :: 'v \text{ propo}$  and ca ::  $'v \text{ connective}$  and  $\chi s \chi s' :: 'v \text{ propo list}$ 
assume corr-ca: wf-conn ca ( $\chi s @ \varphi \# \chi s'$ )
and simple-not: simple (conn ca ( $\chi s @ \varphi \# \chi s'$ ))
hence False

```

**proof** (*case-tac* ca *rule: connective-cases-arity*)

```

fix x :: 'v
assume simple (conn ca ( $\chi s @ \varphi \# \chi s'$ )) and  $ca = CT \vee ca = CF \vee ca = CVar x$ 
hence  $\chi s @ \varphi \# \chi s' = []$  using corr-ca by auto
thus False by auto

```

**next**

```

assume simple: simple (conn ca ( $\chi s @ \varphi \# \chi s'$ ))
and ca:  $ca \in \text{binary-connectives}$ 
obtain a b where  $\chi s @ \varphi \# \chi s' = [a, b]$ 
    using corr-ca ca list-length2-decomp wf-conn-bin-list-length
    by (metis append-assoc length-Cons length-append length-append-singleton)
thus False using simple ca ab conn.simps(5,6,7,8) unfolding binary-connectives-def by auto
next
assume simple: simple (conn ca ( $\chi s @ \varphi \# \chi s'$ ))

```

```

and ca: ca = CNot
hence empty:  $\chi s = []$   $\chi s' = []$  using corr-ca by auto
thus False using simple ca conn.simps(4) by auto
qed
thus simple (conn ca ( $\chi s @ \varphi' \# \chi s'$ )) by blast
qed

```

**lemma** *push-conn-inside-not-true-false*:  
*push-conn-inside c c'  $\varphi \psi \implies \psi \neq FT \wedge \psi \neq FF$*   
**by** (induct rule: *push-conn-inside.induct*, auto)

**lemma** *push-conn-inside-inv*:

```

fixes  $\varphi \psi :: 'v$  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** –

```

{
  {
    fix  $\varphi \psi :: 'v$  propo
    have H: push-conn-inside c c'  $\varphi \psi \implies$  all-subformula-st simple-not-symb  $\varphi$ 
       $\implies$  all-subformula-st simple-not-symb  $\psi$ 
      by (induct  $\varphi \psi$  rule: push-conn-inside.induct, auto)
  } note H = this
}

```

```

fix  $\varphi \psi :: 'v$  propo

```

```

have H: propo-rew-step (push-conn-inside c c')  $\varphi \psi \implies$  all-subformula-st simple-not-symb  $\varphi$ 
   $\implies$  all-subformula-st simple-not-symb  $\psi$ 

```

```

apply (induct  $\varphi \psi$  rule: propo-rew-step.induct)

```

```

using H apply simp

```

**proof** (case-tac ca rule: connective-cases-arity)

```

fix  $\varphi \varphi' :: 'v$  propo and c:: 'v connective and  $\xi \xi':: 'v$  propo list
and x:: 'v

```

```

assume wf-conn c ( $\xi @ \varphi \# \xi'$ )

```

```

and c = CT  $\vee$  c = CF  $\vee$  c = CVar x

```

```

hence  $\xi @ \varphi \# \xi' = []$  by auto

```

```

hence False by auto

```

```

thus all-subformula-st simple-not-symb (conn c ( $\xi @ \varphi' \# \xi'$ )) by blast

```

**next**

```

fix  $\varphi \varphi' :: 'v$  propo and ca:: 'v connective and  $\xi \xi':: 'v$  propo list

```

```

and x:: 'v

```

```

assume rel: propo-rew-step (push-conn-inside c c')  $\varphi \varphi'$ 

```

```

and  $\varphi\text{-}\varphi'$ : all-subformula-st simple-not-symb  $\varphi \implies$  all-subformula-st simple-not-symb  $\varphi'$ 

```

```

and corr: wf-conn ca ( $\xi @ \varphi \# \xi'$ )

```

```

and n: all-subformula-st simple-not-symb (conn ca ( $\xi @ \varphi \# \xi'$ ))

```

```

and c: ca = CNot

```

```

have empty:  $\xi = []$   $\xi' = []$  using c corr by auto

```

```

hence simple-not:all-subformula-st simple-not-symb (FNot  $\varphi$ ) using corr c n by auto

```

```

hence simple  $\varphi$ 

```

```

using all-subformula-st-test-symb-true-phi simple-not-symb.simps(1) by blast

```

```

hence simple  $\varphi'$ 

```

```

using rel simple-propo-rew-step-push-conn-inside-inv by blast

```

```

thus all-subformula-st simple-not-symb (conn ca ( $\xi @ \varphi' \# \xi'$ )) using c empty

```

```

by (metis simple-not  $\varphi\text{-}\varphi'$  append-Nil conn.simps(4) all-subformula-st-decomp-explicit(3))

```



```

    simple-not-symb.simps(1))
next
  fix  $\varphi \varphi' :: 'v$  propo and  $ca :: 'v$  connective and  $\xi \xi' :: 'v$  propo list
  and  $x :: 'v$ 
  assume rel: propo-rew-step (push-conn-inside  $c \ c'$ )  $\varphi \varphi'$ 
  and  $n\varphi$ : all-subformula-st simple-not-symb  $\varphi \implies$  all-subformula-st simple-not-symb  $\varphi'$ 
  and corr: wf-conn  $ca (\xi @ \varphi \# \xi')$ 
  and  $n$ : all-subformula-st simple-not-symb (conn  $ca (\xi @ \varphi \# \xi')$ )
  and  $c$ :  $ca \in$  binary-connectives

  have all-subformula-st simple-not-symb  $\varphi$ 
    using  $n \ c \ corr$  all-subformula-st-decomp by fastforce
  hence  $\varphi'$ : all-subformula-st simple-not-symb  $\varphi'$  using  $n\varphi$  by blast
  obtain  $a \ b$  where  $ab$ :  $[a, b] = (\xi @ \varphi \# \xi')$ 
    using corr  $c$  list-length2-decomp wf-conn-bin-list-length by metis
  hence  $\xi @ \varphi' \# \xi' = [a, \varphi'] \vee (\xi @ \varphi' \# \xi') = [\varphi', b]$ 
    using  $ab$  by (metis (no-types, hide-lams) append-Cons append-Nil append-Nil2
      append-is-Nil-conv butlast.simps(2) butlast-append list.sel(3) tl-append2)
  moreover
  {
    fix  $\chi :: 'v$  propo
    have  $wf'$ : wf-conn  $ca [a, b]$ 
      using  $ab \ corr$  by presburger
    have all-subformula-st simple-not-symb (conn  $ca [a, b]$ )
      using  $ab \ n$  by presburger
    hence all-subformula-st simple-not-symb  $\chi \vee \chi \notin$  set  $(\xi @ \varphi' \# \xi')$ 
      using  $wf'$  by (metis (no-types)  $\varphi'$  all-subformula-st-decomp calculation insert-iff
        list.set(2))
  }
  hence  $\forall \varphi. \varphi \in$  set  $(\xi @ \varphi' \# \xi') \longrightarrow$  all-subformula-st simple-not-symb  $\varphi$ 
    by (metis (no-types))

  moreover have simple-not-symb (conn  $ca (\xi @ \varphi' \# \xi')$ )
    using  $ab$  conn-inj-not(1) corr wf-conn-list-decomp(4) wf-conn-no-arity-change
      not-Cons-self2 self-append-conv2 simple-not-symb.elims(3) by (metis (no-types)  $c$ 
      calculation(1) wf-conn-binary)
  moreover have wf-conn  $ca (\xi @ \varphi' \# \xi')$  using  $c$  calculation(1) by auto
  ultimately show all-subformula-st simple-not-symb (conn  $ca (\xi @ \varphi' \# \xi')$ )
    by (metis all-subformula-st-decomp-imp)
qed
}
moreover {
  fix  $ca :: 'v$  connective and  $\xi \xi' :: 'v$  propo list and  $\varphi \varphi' :: 'v$  propo
  have propo-rew-step (push-conn-inside  $c \ c'$ )  $\varphi \varphi' \implies$  wf-conn  $ca (\xi @ \varphi \# \xi')$ 
     $\implies$  simple-not-symb (conn  $ca (\xi @ \varphi \# \xi')$ )  $\implies$  simple-not-symb  $\varphi'$ 
     $\implies$  simple-not-symb (conn  $ca (\xi @ \varphi' \# \xi')$ )
  by (metis append-self-conv2 conn.simps(4) conn-inj-not(1) simple-not-symb.elims(3)
    simple-not-symb.simps(1) simple-propo-rew-step-push-conn-inside-inv
    wf-conn-no-arity-change-helper wf-conn-list-decomp(4) wf-conn-no-arity-change)
}
ultimately show simple-not  $\psi$ 
  using full-propo-rew-step-inv-stay'[of push-conn-inside  $c \ c'$  simple-not-symb] assms
  unfolding no-T-F-except-top-level-def simple-not-def full-unfold by metis
next
{

```

```

fix  $\varphi \psi :: 'v \text{ propo}$ 
have  $H: \text{propo-rew-step } (\text{push-conn-inside } c \ c') \ \varphi \ \psi \implies \text{no-T-F-except-top-level } \varphi$ 
 $\implies \text{no-T-F-except-top-level } \psi$ 
proof -
  assume  $\text{rel: propo-rew-step } (\text{push-conn-inside } c \ c') \ \varphi \ \psi$ 
  and  $\text{no-T-F-except-top-level } \varphi$ 
  hence  $\text{no-T-F } \varphi \vee \varphi = FF \vee \varphi = FT$ 
    by (metis  $\text{no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb}$ )
  moreover {
    assume  $\varphi = FF \vee \varphi = FT$ 
    hence  $\text{False}$  using  $\text{rel propo-rew-step-push-conn-inside}$  by blast
    hence  $\text{no-T-F-except-top-level } \psi$  by blast
  }
  moreover {
    assume  $\text{no-T-F } \varphi \wedge \varphi \neq FF \wedge \varphi \neq FT$ 
    hence  $\text{no-T-F } \psi$  using  $\text{rel push-conn-insidec-in-c'-symb-no-T-F}$  by blast
    hence  $\text{no-T-F-except-top-level } \psi$  using  $\text{no-T-F-no-T-F-except-top-level}$  by blast
  }
  ultimately show  $\text{no-T-F-except-top-level } \psi$  by blast
qed
}
moreover {
  fix  $ca :: 'v \text{ connective}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $\varphi \varphi' :: 'v \text{ propo}$ 
  assume  $\text{rel: propo-rew-step } (\text{push-conn-inside } c \ c') \ \varphi \ \varphi'$ 
  assume  $\text{corr: wf-conn } ca \ (\xi @ \varphi \# \xi')$ 
  hence  $c: ca \neq CT \wedge ca \neq CF$  by auto
  assume  $\text{no-T-F: no-T-F-symb-except-toplevel } (\text{conn } ca \ (\xi @ \varphi \# \xi'))$ 
  have  $\text{no-T-F-symb-except-toplevel } (\text{conn } ca \ (\xi @ \varphi' \# \xi'))$ 
  proof
    have  $c: ca \neq CT \wedge ca \neq CF$  using  $\text{corr}$  by auto
    have  $\zeta: \forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \ \zeta \neq FT \wedge \zeta \neq FF$ 
      using  $\text{corr no-T-F no-T-F-symb-except-toplevel-if-is-a-true-false}$  by blast
    hence  $\varphi \neq FT \wedge \varphi \neq FF$  by auto
    from  $\text{rel this}$  have  $\varphi' \neq FT \wedge \varphi' \neq FF$ 
      apply (induct rule:  $\text{propo-rew-step.induct}$ )
      by (metis  $\text{append-is-Nil-conv conn.simps(2) conn-inj list.distinct(1)}$ 
         $\text{wf-conn-helper-facts(3) wf-conn-list(1) wf-conn-no-arity-change}$ 
         $\text{wf-conn-no-arity-change-helper push-conn-inside-not-true-false}$ )
    hence  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \ \zeta \neq FT \wedge \zeta \neq FF$  using  $\zeta$  by auto
    moreover have  $\text{wf-conn } ca \ (\xi @ \varphi' \# \xi')$ 
      using  $\text{corr wf-conn-no-arity-change}$  by (metis  $\text{wf-conn-no-arity-change-helper}$ )
    ultimately show  $\text{no-T-F-symb } (\text{conn } ca \ (\xi @ \varphi' \# \xi'))$  using  $\text{no-T-F-symb.intros } c$  by metis
  qed
}
ultimately show  $\text{no-T-F-except-top-level } \psi$ 
  using  $\text{full-propo-rew-step-inv-stay'[of push-conn-inside } c \ c' \ \text{no-T-F-symb-except-toplevel}]$ 
  assms  $\text{unfolding no-T-F-except-top-level-def full-unfold}$  by metis

next
{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have  $H: \text{push-conn-inside } c \ c' \ \varphi \ \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$ 
    by (induct  $\varphi \ \psi$  rule:  $\text{push-conn-inside.induct, auto}$ )
}
thus  $\text{no-equiv } \psi$ 

```

**using** *full-propo-rew-step-inv-stay-conn*[of *push-conn-inside* *c c'* *no-equiv-symb*] *assms*  
*no-equiv-symb-conn-characterization* **unfolding** *no-equiv-def* **by** *metis*

**next**

```
{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have  $H: \text{push-conn-inside } c \ c' \ \varphi \ \psi \implies \text{no-imp } \varphi \implies \text{no-imp } \psi$ 
    by (induct  $\varphi \ \psi$  rule: push-conn-inside.induct, auto)
}
thus no-imp  $\psi$ 
  using full-propo-rew-step-inv-stay-conn[of push-conn-inside c c' no-imp-symb] assms
  no-imp-symb-conn-characterization unfolding no-imp-def by metis
qed
```

**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  $c \ c' \ \psi$ 
using c-in-c'-symb-rew assms full-propo-rew-step-subformula by blast
```

### 8.5.1 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*  $\varphi \implies \text{only-c-inside-symb } c \ \varphi$  |  
*simple-cnot-only-c-inside*[*simp*]: *simple*  $\varphi \implies \text{only-c-inside-symb } c \ (FNot \ \varphi)$  |  
*only-c-inside-into-only-c-inside*: *wf-conn*  $c \ l \implies \text{only-c-inside-symb } c \ (\text{conn } c \ l)$

**lemma** *only-c-inside-symb-simp*[*simp*]:

*only-c-inside-symb*  $c \ FF$  *only-c-inside-symb*  $c \ FT$  *only-c-inside-symb*  $c \ (FVar \ x)$  **by** *auto*

**definition** *only-c-inside* **where** *only-c-inside*  $c = \text{all-subformula-st } (\text{only-c-inside-symb } c)$

**lemma** *only-c-inside-symb-decomp*:

```
only-c-inside-symb  $c \ \psi \longleftrightarrow (\text{simple } \psi$ 
   $\vee (\exists \varphi'. \ \psi = FNot \ \varphi' \wedge \text{simple } \varphi')$ 
   $\vee (\exists l. \ \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l))$ 
by (auto simp add: only-c-inside-symb.intros(3)) (induct rule: only-c-inside-symb.induct, auto)
```

**lemma** *only-c-inside-symb-decomp-not*[*simp*]:

```
fixes  $c :: 'v \text{ connective}$ 
assumes  $c: c \neq CNot$ 
shows only-c-inside-symb  $c \ (FNot \ \psi) \longleftrightarrow \text{simple } \psi$ 
apply (auto simp add: only-c-inside-symb.intros(3))
by (induct FNot  $\psi$  rule: only-c-inside-symb.induct, auto simp add: wf-conn-list(8)  $c$ )
```

**lemma** *only-c-inside-decomp-not*[*simp*]:

**assumes**  $c: c \neq CNot$   
**shows**  $only\text{-}c\text{-}inside\ c\ (FNot\ \psi) \longleftrightarrow simple\ \psi$   
**by** (*metis* (*no-types*, *hide-lams*) *all-subformula-st-def all-subformula-st-test-symb-true-phi c*  
*only-c-inside-def only-c-inside-symb-decomp-not simple-only-c-inside*  
*subformula-conn-decomp-simple*)

**lemma** *only-c-inside-decomp*:

*only-c-inside*  $c\ \varphi \longleftrightarrow$   
 $(\forall \psi. \psi \preceq \varphi \longrightarrow (simple\ \psi \vee (\exists \varphi'. \psi = FNot\ \varphi' \wedge simple\ \varphi') \vee (\exists l. \psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l)))$   
**unfolding** *only-c-inside-def* **by** (*auto simp add: all-subformula-st-def only-c-inside-symb-decomp*)

**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*:  $wf\text{-}conn\ c'\ l$   
**shows** *False*

**proof** –

**let**  $? \psi = conn\ c'\ l$   
**have**  $simple\ ? \psi \vee (\exists \varphi'. ? \psi = FNot\ \varphi' \wedge simple\ \varphi') \vee (\exists l. ? \psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l)$   
**using** *only-c-inside-decomp only incl* **by** *blast*  
**moreover** **have**  $\neg simple\ ? \psi$   
**using** *wf simple-decomp* **by** (*metis*  $c'$  *connective.distinct(19) connective.distinct(7,9,21,29,31)*  
*wf-conn-list(1-3)*)  
**moreover**  
 $\{$   
 $\quad$  **fix**  $\varphi'$   
 $\quad$  **have**  $? \psi \neq FNot\ \varphi'$  **using**  $c'$  *conn-inj-not(1) wf* **by** *blast*  
 $\}$   
**ultimately obtain**  $l :: 'v\ propo\ list$  **where**  $? \psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l$  **by** *metis*  
**hence**  $c = c'$  **using** *conn-inj wf* **by** *metis*  
**thus** *False* **using**  $cc'$  **by** *auto*

**qed**

**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**  $only\text{-}c\text{-}inside\ c\ \varphi \implies c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ \varphi$   
**apply** (*rule ccontr*)  
**apply** (*cases rule: not-c-in-c'-symb.cases, auto*)  
**by** (*metis*  $\delta\ c\ c'$  *connective.distinct(37,39) list.distinct(1) only-c-inside-c-c'-false*  
*subformula-in-binary-conn(1,2) wf-conn.simps*) $+$

**lemma** *c-in-c'-symb-decomp-level1*:

**fixes**  $l :: 'v\ propo\ list$  **and**  $c\ c'\ ca :: 'v\ connective$   
**shows**  $wf\text{-}conn\ ca\ l \implies ca \neq c \implies c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ (conn\ ca\ l)$

**proof** –

**have**  $not\text{-}c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ (conn\ ca\ l) \implies wf\text{-}conn\ ca\ l \implies ca = c$   
**by** (*induct conn ca l rule: not-c-in-c'-symb.induct, auto simp add: conn-inj*)  
**thus**  $wf\text{-}conn\ ca\ l \implies ca \neq c \implies c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ (conn\ ca\ l)$  **by** *blast*

**qed**

**lemma** *only-c-inside-implies-c-in-c'-only*:

**assumes**  $\delta: c \neq c'$  **and**  $c: c = CAnd \vee c = COr$  **and**  $c': c' = CAnd \vee c' = COr$

shows *only-c-inside*  $c \varphi \implies c\text{-in-}c'\text{-only } c \ c' \ \varphi$   
**unfolding** *c-in-c'-only-def* *all-subformula-st-def*  
**using** *only-c-inside-implies-c-in-c'-symb*  
 by (*metis* *all-subformula-st-def* *assms*(1)  $c \ c'$  *only-c-inside-def* *subformula-trans*)

**lemma** *c-in-c'-symb-c-implies-only-c-inside*:

**assumes**  $\delta$ :  $c = CAnd \vee c = COr \ c' = CAnd \vee c' = COr \ c \neq c'$  **and** *wf*: *wf-conn*  $c \ [\varphi, \psi]$   
**and** *inv*: *no-equiv* (*conn*  $c \ l$ ) *no-imp* (*conn*  $c \ l$ ) *simple-not* (*conn*  $c \ l$ )  
**shows** *wf-conn*  $c \ l \implies c\text{-in-}c'\text{-only } c \ c' \ (\text{conn } c \ l) \implies (\forall \psi \in \text{set } l. \text{only-c-inside } c \ \psi)$

**using** *inv*

**proof** (*induct* *conn*  $c \ l$  *arbitrary*:  $l$  *rule*: *propo-induct-arity*)

**case** (*nullary*  $x$ )

**thus** ?*case* **by** (*auto* *simp* *add*: *wf-conn-list* *assms*)

**next**

**case** (*unary*  $\varphi \ la$ )

**hence**  $c = CNot \wedge la = [\varphi]$  **by** (*metis* (*no-types*) *wf-conn-list*(8))

**thus** ?*case* **using** *assms*(2) *assms*(1) **by** *blast*

**next**

**case** (*binary*  $\varphi1 \ \varphi2$ )

**note**  $IH\varphi1 = \text{this}(1)$  **and**  $IH\varphi2 = \text{this}(2)$  **and**  $\varphi = \text{this}(3)$  **and** *only* = *this*(5) **and** *wf* = *this*(4)  
**and** *no-equiv* = *this*(6) **and** *no-imp* = *this*(7) **and** *simple-not* = *this*(8)

**hence**  $l: l = [\varphi1, \varphi2]$  **by** (*meson* *wf-conn-list*(4-7))

**let** ? $\varphi = \text{conn } c \ l$

**obtain**  $c1 \ l1 \ c2 \ l2$  **where**  $\varphi1: \varphi1 = \text{conn } c1 \ l1$  **and**  $\text{wf}\varphi1: \text{wf-conn } c1 \ l1$

**and**  $\varphi2: \varphi2 = \text{conn } c2 \ l2$  **and**  $\text{wf}\varphi2: \text{wf-conn } c2 \ l2$  **using** *exists-c-conn* **by** *metis*

**hence** *c-in-only* $\varphi1: c\text{-in-}c'\text{-only } c \ c' \ (\text{conn } c1 \ l1)$  **and** *c-in-c'-only*  $c \ c' \ (\text{conn } c2 \ l2)$

**using** *only*  $l$  **unfolding** *c-in-c'-only-def* **using** *assms*(1) **by** *auto*

**have** *inc* $\varphi1: \varphi1 \preceq ?\varphi$  **and** *inc* $\varphi2: \varphi2 \preceq ?\varphi$

**using**  $\varphi1 \ \varphi2 \ \varphi$  *local.wf* **by** (*metis* *conn.simps*(5-8) *helper-fact* *subformula-in-binary-conn*(1,2))+

**have** *c1-eq*:  $c1 \neq CEq$  **and** *c2-eq*:  $c2 \neq CEq$

**unfolding** *no-equiv-def* **using** *inc* $\varphi1$  *inc* $\varphi2$  **by** (*metis*  $\varphi1 \ \varphi2 \ \text{wf}\varphi1 \ \text{wf}\varphi2$  *assms*(1) *no-equiv*  
*no-equiv-eq*(1) *no-equiv-symb.elims*(3) *no-equiv-symb-conn-characterization* *wf-conn-list*(4,5)  
*no-equiv-def* *subformula-all-subformula-st*)+

**have** *c1-imp*:  $c1 \neq CImp$  **and** *c2-imp*:  $c2 \neq CImp$

**using** *no-imp* **by** (*metis*  $\varphi1 \ \varphi2$  *all-subformula-st-decomp-explicit-imp*(2,3) *assms*(1)  
*conn.simps*(5,6)  $l$  *no-imp-imp*(1) *no-imp-symb.elims*(3) *no-imp-symb-conn-characterization*  
*wf* $\varphi1 \ \text{wf}\varphi2$  *all-subformula-st-decomp* *no-imp-symb-conn-characterization*)+

**have** *c1c*:  $c1 \neq c'$

**proof**

**assume** *c1c*:  $c1 = c'$

**then obtain**  $\xi1 \ \xi2$  **where**  $l1: l1 = [\xi1, \xi2]$

**by** (*metis* *assms*(2) *connective.distinct*(37,39) *helper-fact* *wf* $\varphi1$  *wf-conn.simps*  
*wf-conn-list-decomp*(1-3))

**have** *c-in-c'-only*  $c \ c' \ (\text{conn } c \ [\text{conn } c' \ l1, \varphi2])$  **using** *c1c*  $l$  *only*  $\varphi1$  **by** *auto*

**moreover have** *not-c-in-c'-symb*  $c \ c' \ (\text{conn } c \ [\text{conn } c' \ l1, \varphi2])$

**using**  $l1 \ \varphi1 \ c1c \ l$  *local.wf* *not-c-in-c'-symb-l* *wf* $\varphi1$  **by** *blast*

**ultimately show** *False* **using**  $\varphi1 \ c1c \ l \ l1$  *local.wf* *not-c-in-c'-simp*(4) *wf* $\varphi1$  **by** *blast*

**qed**

**hence**  $(\varphi1 = \text{conn } c \ l1 \wedge \text{wf-conn } c \ l1) \vee (\exists \psi1. \varphi1 = FNot \ \psi1) \vee \text{simple } \varphi1$

**by** (*metis*  $\varphi1$  *assms*(1-3) *c1-eq* *c1-imp* *simple.elims*(3) *wf* $\varphi1$  *wf-conn-list*(4) *wf-conn-list*(5-7))

**moreover** {

**assume**  $\varphi1 = \text{conn } c \ l1 \wedge \text{wf-conn } c \ l1$

```

hence only-c-inside  $c \varphi 1$ 
  by (metis IH $\varphi 1$   $\varphi 1$  all-subformula-st-decomp-imp inc $\varphi 1$  no-equiv no-equiv-def no-imp no-imp-def
    c-in-only $\varphi 1$  only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
    subformula-all-subformula-st)
}
moreover {
  assume  $\exists \psi 1. \varphi 1 = FNot \psi 1$ 
  then obtain  $\psi 1$  where  $\varphi 1 = FNot \psi 1$  by metis
  hence only-c-inside  $c \varphi 1$ 
    by (metis all-subformula-st-def assms(1) connective.distinct(37,39) inc $\varphi 1$ 
      only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
}
moreover {
  assume simple  $\varphi 1$ 
  hence only-c-inside  $c \varphi 1$ 
    by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
      only-c-inside-decomp-not only-c-inside-def)
}
ultimately have only-c-inside $\varphi 1$ : only-c-inside  $c \varphi 1$  by metis

have c-in-only $\varphi 2$ : c-in-c'-only  $c c'$  (conn  $c2 l2$ )
  using only  $l \varphi 2$  wf $\varphi 2$  assms unfolding c-in-c'-only-def by auto
have  $c2c$ :  $c2 \neq c'$ 
proof
  assume  $c2c$ :  $c2 = c'$ 
  then obtain  $\xi 1 \xi 2$  where  $l2 = [\xi 1, \xi 2]$ 
    by (metis assms(2) wf $\varphi 2$  wf-conn.simps connective.distinct(7,9,19,21,29,31,37,39))
  hence c-in-c'-symb  $c c'$  (conn  $c [\varphi 1, \text{conn } c' l2]$ )
    using  $c2c l$  only  $\varphi 2$  all-subformula-st-test-symb-true-phi unfolding c-in-c'-only-def by auto
  moreover have not-c-in-c'-symb  $c c'$  (conn  $c [\varphi 1, \text{conn } c' l2]$ )
    using assms(1)  $c2c l2$  not-c-in-c'-symb-r wf $\varphi 2$  wf-conn-helper-facts(5,6) by metis
  ultimately show False by auto
qed
hence  $(\varphi 2 = \text{conn } c l2 \wedge \text{wf-conn } c l2) \vee (\exists \psi 2. \varphi 2 = FNot \psi 2) \vee \text{simple } \varphi 2$ 
  using  $c2\text{-eq}$  by (metis  $\varphi 2$  assms(1-3)  $c2\text{-eq}$   $c2\text{-imp}$  simple.elims(3) wf $\varphi 2$  wf-conn-list(4-7))
moreover {
  assume  $\varphi 2 = \text{conn } c l2 \wedge \text{wf-conn } c l2$ 
  hence only-c-inside  $c \varphi 2$ 
    by (metis IH $\varphi 2$   $\varphi 2$  all-subformula-st-decomp inc $\varphi 2$  no-equiv no-equiv-def no-imp no-imp-def
      c-in-only $\varphi 2$  only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
      subformula-all-subformula-st)
}
moreover {
  assume  $\exists \psi 2. \varphi 2 = FNot \psi 2$ 
  then obtain  $\psi 2$  where  $\varphi 2 = FNot \psi 2$  by metis
  hence only-c-inside  $c \varphi 2$ 
    by (metis all-subformula-st-def assms(1-3) connective.distinct(38,40) inc $\varphi 2$ 
      only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
}
moreover {
  assume simple  $\varphi 2$ 
  hence only-c-inside  $c \varphi 2$ 
    by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
      only-c-inside-decomp-not only-c-inside-def)
}

```

ultimately have *only-c-inside* $\varphi_2$ : *only-c-inside*  $c$   $\varphi_2$  by *metis*  
 show ?case using *l only-c-inside* $\varphi_1$  *only-c-inside* $\varphi_2$  by *auto*  
 qed

### 8.5.2 Push Conjunction

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

**lemma** *pushConj-consistent*: *preserves-un-sat pushConj*  
*unfolding pushConj-def* by (*simp add: push-conn-inside-consistent*)

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

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

**lemma** *pushConj-inv*:  
*fixes*  $\varphi \psi :: 'v \text{ propo}$   
*assumes* *full* (*propo-rew-step 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*  $\psi$   
*using push-conn-inside-inv* *assms* *unfolding pushConj-def* by *metis+*

**lemma** *pushConj-full-propo-rew-step*:  
*fixes*  $\varphi \psi :: 'v \text{ propo}$   
*assumes*  
*no-equiv*  $\varphi$  *and*  
*no-imp*  $\varphi$  *and*  
*full* (*propo-rew-step pushConj*)  $\varphi \psi$  *and*  
*no-T-F-except-top-level*  $\varphi$  *and*  
*simple-not*  $\varphi$   
*shows* *and-in-or-only*  $\psi$   
*using* *assms push-conn-inside-full-propo-rew-step*  
*unfolding pushConj-def and-in-or-only-def c-in-c'-only-def* by (*metis (no-types)*)

### 8.5.3 Push Disjunction

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

**lemma** *pushDisj-consistent*: *preserves-un-sat pushDisj*  
*unfolding pushDisj-def* by (*simp add: push-conn-inside-consistent*)

**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*]:  
 $\sim \text{or-in-and-only } (FOr (FAnd \psi_1 \psi_2) \varphi')$   
*unfolding or-in-and-only-def*  
*by* (*metis all-subformula-st-test-symb-true-phi conn.simps(5-6) not-c-in-c'-symb-l*  
*wf-conn-helper-facts(5) wf-conn-helper-facts(6)*)

**lemma** *pushDisj-inv*:

**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes**  $\text{full } (\text{propo-rew-step } \text{pushDisj}) \varphi \psi$   
**and**  $\text{no-equiv } \varphi$  **and**  $\text{no-imp } \varphi$  **and**  $\text{no-T-F-except-top-level } \varphi$  **and**  $\text{simple-not } \varphi$   
**shows**  $\text{no-equiv } \psi$  **and**  $\text{no-imp } \psi$  **and**  $\text{no-T-F-except-top-level } \psi$  **and**  $\text{simple-not } \psi$   
**using**  $\text{push-conn-inside-inv } \text{assms}$  **unfolding**  $\text{pushDisj-def}$  **by**  $\text{metis+}$

**lemma**  $\text{pushDisj-full-propo-rew-step}$ :

**fixes**  $\varphi \psi :: 'v \text{ propo}$   
**assumes**  
 $\text{no-equiv } \varphi$  **and**  
 $\text{no-imp } \varphi$  **and**  
 $\text{full } (\text{propo-rew-step } \text{pushDisj}) \varphi \psi$  **and**  
 $\text{no-T-F-except-top-level } \varphi$  **and**  
 $\text{simple-not } \varphi$   
**shows**  $\text{or-in-and-only } \psi$   
**using**  $\text{assms } \text{push-conn-inside-full-propo-rew-step}$   
**unfolding**  $\text{pushDisj-def}$   $\text{or-in-and-only-def}$   $\text{c-in-c'-only-def}$  **by**  $(\text{metis } (\text{no-types}))$

## 9 The full transformations

### 9.1 Abstract Property characterizing that only some connective are inside the others

#### 9.1.1 Definition

The normal is a super group of groups

**inductive**  $\text{grouped-by} :: 'a \text{ connective} \Rightarrow 'a \text{ propo} \Rightarrow \text{bool}$  **for**  $c$  **where**  
 $\text{simple-is-grouped}[\text{simp}]: \text{simple } \varphi \Longrightarrow \text{grouped-by } c \varphi \mid$   
 $\text{simple-not-is-grouped}[\text{simp}]: \text{simple } \varphi \Longrightarrow \text{grouped-by } c (\text{FNot } \varphi) \mid$   
 $\text{connected-is-group}[\text{simp}]: \text{grouped-by } c \varphi \Longrightarrow \text{grouped-by } c \psi \Longrightarrow \text{wf-conn } c [\varphi, \psi]$   
 $\Longrightarrow \text{grouped-by } c (\text{conn } c [\varphi, \psi])$

**lemma**  $\text{simple-clause}[\text{simp}]$ :

$\text{grouped-by } c \text{ FT}$   
 $\text{grouped-by } c \text{ FF}$   
 $\text{grouped-by } c (\text{FVar } x)$   
 $\text{grouped-by } c (\text{FNot } \text{FT})$   
 $\text{grouped-by } c (\text{FNot } \text{FF})$   
 $\text{grouped-by } c (\text{FNot } (\text{FVar } x))$   
**by**  $\text{simp+}$

**lemma**  $\text{only-c-inside-symb-c-eq-c'}$ :

$\text{only-c-inside-symb } c (\text{conn } c' [\varphi 1, \varphi 2]) \Longrightarrow c' = \text{CAnd} \vee c' = \text{COr} \Longrightarrow \text{wf-conn } c' [\varphi 1, \varphi 2]$   
 $\Longrightarrow c' = c$   
**by**  $(\text{induct } \text{conn } c' [\varphi 1, \varphi 2] \text{ rule: } \text{only-c-inside-symb.induct, auto simp add: conn-inj})$

**lemma**  $\text{only-c-inside-c-eq-c'}$ :

$\text{only-c-inside } c (\text{conn } c' [\varphi 1, \varphi 2]) \Longrightarrow c' = \text{CAnd} \vee c' = \text{COr} \Longrightarrow \text{wf-conn } c' [\varphi 1, \varphi 2] \Longrightarrow c = c'$   
**unfolding**  $\text{only-c-inside-def}$   $\text{all-subformula-st-def}$  **using**  $\text{only-c-inside-symb-c-eq-c'}$   $\text{subformula-refl}$   
**by**  $\text{blast}$

**lemma**  $\text{only-c-inside-imp-grouped-by}$ :

**assumes**  $c: c \neq \text{CNot}$  **and**  $c': c' = \text{CAnd} \vee c' = \text{COr}$   
**shows**  $\text{only-c-inside } c \varphi \Longrightarrow \text{grouped-by } c \varphi$  **(is**  $?O \varphi \Longrightarrow ?G \varphi)$



```

proof (induct  $\varphi$  rule: propo-induct-arity)
  case (nullary  $\varphi$   $x$ )
  thus ?G  $\varphi$  by auto
next
  case (unary  $\psi$ )
  thus ?G (FNot  $\psi$ ) by (auto simp add: c)
next
  case (binary  $\varphi$   $\varphi 1$   $\varphi 2$ )
  note IH $\varphi 1 = \text{this}(1)$  and IH $\varphi 2 = \text{this}(2)$  and  $\varphi = \text{this}(3)$  and only =  $\text{this}(4)$ 
  have  $\varphi\text{-conn}$ :  $\varphi = \text{conn } c [\varphi 1, \varphi 2]$  and wf: wf-conn  $c [\varphi 1, \varphi 2]$ 
  proof -
    obtain  $c'' l''$  where  $\varphi\text{-c''}$ :  $\varphi = \text{conn } c'' l''$  and wf: wf-conn  $c'' l''$ 
    using exists-c-conn by metis
    hence  $l''$ :  $l'' = [\varphi 1, \varphi 2]$  using  $\varphi$  by (metis wf-conn-list(4-7))
    have only-c-inside-symb  $c (\text{conn } c'' [\varphi 1, \varphi 2])$ 
    using only all-subformula-st-test-symb-true-phi
    unfolding only-c-inside-def  $\varphi\text{-c'' } l''$  by metis
    hence  $c = c''$ 
    by (metis  $\varphi$   $\varphi\text{-c''}$  conn-inj conn-inj-not(2)  $l''$  list.distinct(1) list.inject wf
      only-c-inside-symb.cases simple.simps(5-8))
    thus  $\varphi = \text{conn } c [\varphi 1, \varphi 2]$  and wf-conn  $c [\varphi 1, \varphi 2]$  using  $\varphi\text{-c''}$  wf  $l''$  by auto
  qed
  have grouped-by  $c \varphi 1$  using wf IH $\varphi 1$  IH $\varphi 2$   $\varphi\text{-conn}$  only  $\varphi$  unfolding only-c-inside-def by auto
  moreover have grouped-by  $c \varphi 2$ 
  using wf  $\varphi$  IH $\varphi 1$  IH $\varphi 2$   $\varphi\text{-conn}$  only unfolding only-c-inside-def by auto
  ultimately show ?G  $\varphi$  using  $\varphi\text{-conn}$  connected-is-group local.wf by blast
qed

```

**lemma** grouped-by-false:

```

grouped-by  $c (\text{conn } c' [\varphi, \psi]) \implies c \neq c' \implies \text{wf-conn } c' [\varphi, \psi] \implies \text{False}$ 
apply (induct conn  $c' [\varphi, \psi]$  rule: grouped-by.induct)
apply (auto simp add: simple-decomp wf-conn-list, auto simp add: conn-inj)
by (metis list.distinct(1) list.sel(3) wf-conn-list(8))+

```

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  $c \varphi \implies \text{super-grouped-by } c c' \varphi$  |  
 connected-is-super-group: super-grouped-by  $c c' \varphi \implies \text{super-grouped-by } c c' \psi \implies \text{wf-conn } c [\varphi, \psi]$   
 $\implies \text{super-grouped-by } c c' (\text{conn } c' [\varphi, \psi])$

**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  $c c' (FNot (FVar x))$ 
by auto

```

**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  $\varphi \implies \text{no-imp } \varphi \implies \text{simple-not } \varphi \implies \text{c-in-c'-only } c c' \varphi$ 

```

```

     $\implies$  super-grouped-by  $c$   $c'$   $\varphi$ 
    (is ?NE  $\varphi \implies$  ?NI  $\varphi \implies$  ?SN  $\varphi \implies$  ?C  $\varphi \implies$  ?S  $\varphi$ )
  proof (induct  $\varphi$  rule: propo-induct-arity)
    case (nullary  $\varphi$   $x$ )
    thus ?S  $\varphi$  by auto
  next
    case (unary  $\varphi$ )
    hence simple-not-symb (FNot  $\varphi$ )
      using all-subformula-st-test-symb-true-phi unfolding simple-not-def by blast
    hence  $\varphi = FT \vee \varphi = FF \vee (\exists x. \varphi = FVar\ x)$  by (case-tac  $\varphi$ , auto)
    thus ?S (FNot  $\varphi$ ) by auto
  next
    case (binary  $\varphi$   $\varphi1$   $\varphi2$ )
    note IH $\varphi1 = \text{this}(1)$  and IH $\varphi2 = \text{this}(2)$  and no-equiv = this(4) and no-imp = this(5)
      and simpleN = this(6) and c-in-c'-only = this(7) and  $\varphi' = \text{this}(3)$ 
    {
      assume  $\varphi = FImp\ \varphi1\ \varphi2 \vee \varphi = FEq\ \varphi1\ \varphi2$ 
      hence False using no-equiv no-imp by auto
      hence ?S  $\varphi$  by auto
    }
    moreover {
      assume  $\varphi: \varphi = \text{conn}\ c' [\varphi1, \varphi2] \wedge \text{wf-conn}\ c' [\varphi1, \varphi2]$ 
      have c-in-c'-only: c-in-c'-only  $c\ c'\ \varphi1 \wedge \text{c-in-c'-only}\ c\ c'\ \varphi2 \wedge \text{c-in-c'-symb}\ c\ c'\ \varphi$ 
        using c-in-c'-only  $\varphi'$  unfolding c-in-c'-only-def by auto
      have super-grouped-by  $c\ c'\ \varphi1$  using  $\varphi\ c'$  no-equiv no-imp simpleN IH $\varphi1$  c-in-c'-only by auto
      moreover have super-grouped-by  $c\ c'\ \varphi2$ 
        using  $\varphi\ c'$  no-equiv no-imp simpleN IH $\varphi2$  c-in-c'-only by auto
      ultimately have ?S  $\varphi$ 
        using super-grouped-by.intros(2)  $\varphi$  by (metis c wf-conn-helper-facts(5,6))
    }
    moreover {
      assume  $\varphi: \varphi = \text{conn}\ c [\varphi1, \varphi2] \wedge \text{wf-conn}\ c [\varphi1, \varphi2]$ 
      hence only-c-inside  $c\ \varphi1 \wedge \text{only-c-inside}\ c\ \varphi2$ 
        using c-in-c'-symb-c-implies-only-c-inside  $c\ c'\ \text{c-in-c'-only}\ \text{list.set-intros}(1)$ 
          wf-conn-helper-facts(5,6) no-equiv no-imp simpleN last-ConsL last-ConsR last-in-set
          list.distinct(1) by (metis (no-types, hide-lams) cc')
      hence only-c-inside  $c\ (\text{conn}\ c [\varphi1, \varphi2])$ 
        unfolding only-c-inside-def using  $\varphi$ 
        by (simp add: only-c-inside-into-only-c-inside all-subformula-st-decomp)
      hence grouped-by  $c\ \varphi$  using  $\varphi$  only-c-inside-imp-grouped-by  $c$  by blast
      hence ?S  $\varphi$  using super-grouped-by.intros(1) by metis
    }
    ultimately show ?S  $\varphi$  by (metis  $\varphi'\ c\ c'\ cc'\ \text{conn.simps}$ (5,6) wf-conn-helper-facts(5,6))
  qed

```

## 9.2 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*  $\varphi \implies \text{no-imp}\ \varphi \implies \text{simple-not}\ \varphi \implies \text{or-in-and-only}\ \varphi \implies \text{is-conj-with-TF}\ \varphi$   
**using** *c-in-c'-only-super-grouped-by*  
**unfolding** *is-conj-with-TF-def or-in-and-only-def c-in-c'-only-def*  
**by** (*simp add: c-in-c'-only-def c-in-c'-only-super-grouped-by*)

**definition** *is-cnf* **where** *is-cnf*  $\varphi$  == *is-conj-with-TF*  $\varphi \wedge \text{no-T-F-except-top-level}\ \varphi$

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

**lemma** *cnf-rew-consistent: preserves-un-sat cnf-rew*  
**by** (simp add: cnf-rew-def elimEquiv-lifted-consistant elim-imp-lifted-consistant elimTB-consistent  
 preserves-un-sat-OO pushDisj-consistent pushNeg-lifted-consistant)

**lemma** *cnf-rew-is-cnf: cnf-rew  $\varphi$   $\varphi' \implies$  is-cnf  $\varphi'$*

**apply** (unfold cnf-rew-def OO-def)

**apply** auto

**proof** –

**fix**  $\varphi$   $\varphiEq$   $\varphiImp$   $\varphiTB$   $\varphiNeg$   $\varphiDisj$  :: 'v propo

**assume** *Eq*: full (propo-rew-step elim-equiv)  $\varphi$   $\varphiEq$

**hence** *no-equiv*: no-equiv  $\varphiEq$  **using** no-equiv-full-propo-rew-step-elim-equiv **by** blast

**assume** *Imp*: full (propo-rew-step elim-imp)  $\varphiEq$   $\varphiImp$

**hence** *no-imp*: no-imp  $\varphiImp$  **using** no-imp-full-propo-rew-step-elim-imp **by** blast

**have** *no-imp-inv*: no-equiv  $\varphiImp$  **using** no-equiv Imp elim-imp-inv **by** blast

**assume** *TB*: full (propo-rew-step elimTB)  $\varphiImp$   $\varphiTB$

**hence** *noTB*: no-T-F-except-top-level  $\varphiTB$

**using** no-imp-inv no-equiv elimTB-full-propo-rew-step **by** blast

**have** *noTB-inv*: no-equiv  $\varphiTB$  no-imp  $\varphiTB$  **using** elimTB-inv TB no-imp no-imp-inv **by** blast+

**assume** *Neg*: full (propo-rew-step pushNeg)  $\varphiTB$   $\varphiNeg$

**hence** *noNeg*: simple-not  $\varphiNeg$

**using** noTB-inv noTB pushNeg-full-propo-rew-step **by** blast

**have** *noNeg-inv*: no-equiv  $\varphiNeg$  no-imp  $\varphiNeg$  no-T-F-except-top-level  $\varphiNeg$

**using** pushNeg-inv Neg noTB noTB-inv **by** blast+

**assume** *Disj*: full (propo-rew-step pushDisj)  $\varphiNeg$   $\varphiDisj$

**hence** *no-Disj*: or-in-and-only  $\varphiDisj$

**using** noNeg-inv noNeg pushDisj-full-propo-rew-step **by** blast

**have** *noDisj-inv*: no-equiv  $\varphiDisj$  no-imp  $\varphiDisj$  no-T-F-except-top-level  $\varphiDisj$

simple-not  $\varphiDisj$

**using** pushDisj-inv Disj noNeg noNeg-inv **by** blast+

**moreover** **have** *is-conj-with-TF*  $\varphiDisj$

**using** or-in-and-only-conjunction-in-disj noDisj-inv no-Disj **by** blast

**ultimately** **show** *is-cnf*  $\varphiDisj$  **unfolding** *is-cnf-def* **by** blast

qed

### 9.3 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**  $\text{no-equiv } \varphi \implies \text{no-imp } \varphi \implies \text{simple-not } \varphi \implies \text{and-in-or-only } \varphi \implies \text{is-disj-with-TF } \varphi$   
**using**  $c\text{-in-}c'\text{-only-super-grouped-by}$   
**unfolding**  $\text{is-disj-with-TF-def}$   $\text{and-in-or-only-def}$   $c\text{-in-}c'\text{-only-def}$   
**by** (*simp add: c-in-c'-only-def c-in-c'-only-super-grouped-by*)

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

### 9.3.1 Full DNF transform

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

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

**lemma**  $\text{dnf-rew-consistent: preserves-un-sat dnf-rew}$   
**by** (*simp add: dnf-rew-def elimEquiv-lifted-consistant elim-imp-lifted-consistant elimTB-consistent preserves-un-sat-OO pushConj-consistent pushNeg-lifted-consistant*)

**theorem**  $\text{dnf-transformation-correction:}$   
 $\text{dnf-rew } \varphi \varphi' \implies \text{is-dnf } \varphi'$   
**apply** (*unfold dnf-rew-def OO-def*)  
**by** (*meson and-in-or-only-conjunction-in-disj elimTB-full-propo-rew-step elimTB-inv(1,2) elim-imp-inv is-dnf-def no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(1-4) pushNeg-full-propo-rew-step pushNeg-inv(1-3)*)

## 10 More aggressive simplifications: Removing true and false at the beginning

### 10.1 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**  $\text{elimTBFull}$  **where**  
 $\text{ElimTBFull1[simp]: elimTBFull (FAnd } \varphi \text{ FT) } \varphi \mid$   
 $\text{ElimTBFull1'[simp]: elimTBFull (FAnd FT } \varphi) \varphi \mid$   
 $\text{ElimTBFull2[simp]: elimTBFull (FAnd } \varphi \text{ FF) FF} \mid$   
 $\text{ElimTBFull2'[simp]: elimTBFull (FAnd FF } \varphi) \text{ FF} \mid$   
 $\text{ElimTBFull3[simp]: elimTBFull (FOr } \varphi \text{ FT) FT} \mid$   
 $\text{ElimTBFull3'[simp]: elimTBFull (FOr FT } \varphi) \text{ FT} \mid$   
 $\text{ElimTBFull4[simp]: elimTBFull (FOr } \varphi \text{ FF) } \varphi \mid$   
 $\text{ElimTBFull4'[simp]: elimTBFull (FOr FF } \varphi) \varphi \mid$   
 $\text{ElimTBFull5[simp]: elimTBFull (FNot FT) FF} \mid$   
 $\text{ElimTBFull5'[simp]: elimTBFull (FNot FF) FT} \mid$

$\text{ElimTBFull6-l[simp]}: \text{elimTBFull } (F\text{Imp } FT \ \varphi) \ \varphi \mid$   
 $\text{ElimTBFull6-l'[simp]}: \text{elimTBFull } (F\text{Imp } FF \ \varphi) \ FT \mid$   
 $\text{ElimTBFull6-r[simp]}: \text{elimTBFull } (F\text{Imp } \varphi \ FT) \ FT \mid$   
 $\text{ElimTBFull6-r'[simp]}: \text{elimTBFull } (F\text{Imp } \varphi \ FF) \ (F\text{Not } \varphi) \mid$

$\text{ElimTBFull7-l[simp]}: \text{elimTBFull } (F\text{Eq } FT \ \varphi) \ \varphi \mid$   
 $\text{ElimTBFull7-l'[simp]}: \text{elimTBFull } (F\text{Eq } FF \ \varphi) \ (F\text{Not } \varphi) \mid$   
 $\text{ElimTBFull7-r[simp]}: \text{elimTBFull } (F\text{Eq } \varphi \ FT) \ \varphi \mid$   
 $\text{ElimTBFull7-r'[simp]}: \text{elimTBFull } (F\text{Eq } \varphi \ FF) \ (F\text{Not } \varphi) \mid$

The transformation is still consistent.

**lemma** *elimTBFull-consistent: preserves-un-sat elimTBFull*

**proof** –

{  
   **fix**  $\varphi \ \psi:: 'b \text{ propo}$   
   **have**  $\text{elimTBFull } \varphi \ \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$   
     **by** (*induct-tac rule: elimTBFull.inducts, auto*)  
 }

**thus** *?thesis* **using** *preserves-un-sat-def* **by** *auto*

**qed**

Contrary to the theorem  $\llbracket \text{no-equiv } ?\varphi; \text{no-imp } ?\varphi; ?\psi \preceq ?\varphi; \neg \text{no-T-F-symb-except-toplevel } ?\psi \rrbracket \implies \exists \psi'. \text{elimTB } ?\psi \ \psi'$ , 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**  $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTBFull } \psi \ \psi'$

**proof** (*induct*  $\psi$  *rule: propo-induct-arity*)

**case** (*nullary*  $\varphi'$ )

**hence** *False* **using** *no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false* **by** *auto*

**thus** *Ex* ( $\text{elimTBFull } \varphi'$ ) **by** *blast*

**next**

**case** (*unary*  $\psi$ )

**hence**  $\psi = FF \vee \psi = FT$  **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*

**thus** *Ex* ( $\text{elimTBFull } (F\text{Not } \psi)$ ) **using** *ElimTBFull5 ElimTBFull5'* **by** *blast*

**next**

**case** (*binary*  $\varphi' \ \psi1 \ \psi2$ )

**hence**  $\psi1 = FT \vee \psi2 = FT \vee \psi1 = FF \vee \psi2 = FF$

**by** (*metis binary-connectives-def conn.simps(5–8) insertI1 insert-commute*  
       *no-T-F-symb-except-toplevel-bin-decom binary.hyps(3)*)

**thus** *Ex* ( $\text{elimTBFull } \varphi'$ ) **using** *elimTBFull.intros binary.hyps(3)* **by** *blast*

**qed**

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**  $\exists \psi \ \psi'. \psi \preceq \varphi \wedge \text{elimTBFull } \psi \ \psi'$

**proof** –

**have** *test-symb-false-nullary*:

$\forall x. \text{no-T-F-symb-except-toplevel } (FF:: 'v \text{ propo}) \wedge \text{no-T-F-symb-except-toplevel } FT$   
      $\wedge \text{no-T-F-symb-except-toplevel } (F\text{Var } (x:: 'v))$

**by** *auto*

**moreover** {

```

fix c :: 'v connective and l :: 'v propo list and ψ :: 'v propo
have H: elimTBFull (conn c l) ψ ⇒ ¬no-T-F-symb-except-toplevel (conn c l)
  by (case-tac (conn c l) rule: elimTBFull.cases, simp-all)
}
ultimately show ?thesis
  using no-test-symb-step-exists[of no-T-F-symb-except-toplevel φ elimTBFull] noTB
  no-T-F-symb-except-toplevel-step-exists' unfolding no-T-F-except-top-level-def by metis
qed

```

```

lemma elimTBFull-full-propo-rew-step:
  fixes φ ψ :: 'v propo
  assumes full (propo-rew-step elimTBFull) φ ψ
  shows no-T-F-except-top-level ψ
  using full-propo-rew-step-subformula no-T-F-except-top-level-rew' assms by fastforce

```

## 10.2 More invariants

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

```

lemma propo-rew-step-ElimEquiv-no-T-F: propo-rew-step elim-equiv φ ψ ⇒ no-T-F φ ⇒ no-T-F ψ
proof (induct rule: propo-rew-step.induct)

```

```

  fix φ' :: 'v propo and ψ' :: 'v propo
  assume a1: no-T-F φ'
  assume a2: elim-equiv φ' ψ'
  have ∀ x0 x1. (¬ elim-equiv (x1 :: 'v propo) x0 ∨ (∃ v2 v3 v4 v5 v6 v7. x1 = FEq v2 v3
    ∧ x0 = FAnd (FImp v4 v5) (FImp v6 v7) ∧ v2 = v4 ∧ v4 = v7 ∧ v3 = v5 ∧ v3 = v6))
    = (¬ elim-equiv x1 x0 ∨ (∃ v2 v3 v4 v5 v6 v7. x1 = FEq v2 v3
    ∧ x0 = FAnd (FImp v4 v5) (FImp v6 v7) ∧ v2 = v4 ∧ v4 = v7 ∧ v3 = v5 ∧ v3 = v6))
  by meson
  hence ∀ p pa. ¬ elim-equiv (p :: 'v propo) pa ∨ (∃ pb pc pd pe pf pg. p = FEq pb pc
    ∧ pa = FAnd (FImp pd pe) (FImp pf pg) ∧ pb = pd ∧ pd = pg ∧ pc = pe ∧ pc = pf)
  using elim-equiv.cases by force
  thus no-T-F ψ' using a1 a2 by fastforce

```

next

```

fix φ φ' :: 'v propo and ξ ξ' :: 'v propo list and c :: 'v connective
assume rel: propo-rew-step elim-equiv φ φ'
and IH: no-T-F φ ⇒ no-T-F φ'
and corr: wf-conn c (ξ @ φ # ξ')
and no-T-F: no-T-F (conn c (ξ @ φ # ξ'))
{
  assume c: c = CNot
  hence empty: ξ = [] ξ' = [] using corr by auto
  hence no-T-F φ using no-T-F c no-T-F-decomp-not by auto
  hence no-T-F (conn c (ξ @ φ' # ξ')) using c empty no-T-F-comp-not IH by auto
}
moreover {
  assume c: c ∈ binary-connectives
  obtain a b where ab: ξ @ φ # ξ' = [a, b]
  using corr c list-length2-decomp wf-conn-bin-list-length by metis
  hence φ: φ = a ∨ φ = b
  by (metis append.simps(1) append-is-Nil-conv list.distinct(1) list.sel(3) nth-Cons-0
    tl-append2)

```

```

have  $\zeta$ :  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{ no-T-F } \zeta$ 
  using no-T-F unfolding no-T-F-def using corr all-subformula-st-decomp by blast

hence  $\varphi'$ : no-T-F  $\varphi'$  using ab IH  $\varphi$  by auto
have  $l'$ :  $\xi @ \varphi' \# \xi' = [\varphi', b] \vee \xi @ \varphi' \# \xi' = [a, \varphi']$ 
  by (metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps(2)
    butlast-append list.distinct(1) list.sel(3))
hence  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \text{ no-T-F } \zeta$  using  $\zeta$   $\varphi'$  ab by fastforce
moreover
  have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \zeta \neq FT \wedge \zeta \neq FF$ 
    using  $\zeta$  corr no-T-F no-T-F-except-top-level-false no-T-F-no-T-F-except-top-level by blast
  hence no-T-F-symb (conn c ( $\xi @ \varphi' \# \xi'$ ))
    by (metis  $\varphi'$   $l'$  ab all-subformula-st-test-symb-true-phi c list.distinct(1)
      list.set-intros(1,2) no-T-F-symb-except-toplevel-bin-decom
      no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(1,2) no-T-F-def wf-conn-binary
      wf-conn-list(1,2))
  ultimately have no-T-F (conn c ( $\xi @ \varphi' \# \xi'$ ))
    by (metis  $l'$  all-subformula-st-decomp-imp c no-T-F-def wf-conn-binary)
}
moreover {
  fix x
  assume  $c = CVar\ x \vee c = CF \vee c = CT$ 
  hence False using corr by auto
  hence no-T-F (conn c ( $\xi @ \varphi' \# \xi'$ )) by auto
}
ultimately show no-T-F (conn c ( $\xi @ \varphi' \# \xi'$ )) using corr wf-conn.cases by metis
qed

```

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 -
{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have propo-rew-step elim-equiv  $\varphi \psi \implies \text{no-T-F-except-top-level } \varphi$ 
     $\implies \text{no-T-F-except-top-level } \psi$ 
  proof -
    assume rel: propo-rew-step elim-equiv  $\varphi \psi$ 
    and no: no-T-F-except-top-level  $\varphi$ 
    {
      assume  $\varphi = FT \vee \varphi = FF$ 
      from rel this have False
      apply (induct rule: propo-rew-step.induct, auto simp add: wf-conn-list(1,2))
      using elim-equiv.simps by blast+
      hence no-T-F-except-top-level  $\psi$  by blast
    }
  moreover {
    assume  $\varphi \neq FT \wedge \varphi \neq FF$ 
    hence no-T-F  $\varphi$  by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
    hence no-T-F  $\psi$  using propo-rew-step-ElimEquiv-no-T-F rel by blast
    hence no-T-F-except-top-level  $\psi$  by (simp add: no-T-F-no-T-F-except-top-level)
  }
  ultimately show no-T-F-except-top-level  $\psi$  by metis
}
qed

```

```

}
moreover {
  fix  $c :: 'v \text{ connective}$  and  $\xi \ \xi' :: 'v \text{ propo list}$  and  $\zeta \ \zeta' :: 'v \text{ propo}$ 
  assume  $\text{rel: propo-rew-step elim-equiv } \zeta \ \zeta'$ 
  and  $\text{incl: } \zeta \preceq \varphi$ 
  and  $\text{corr: wf-conn } c \ (\xi @ \zeta \# \xi')$ 
  and  $\text{no-T-F: no-T-F-symb-except-toplevel } (\text{conn } c \ (\xi @ \zeta \# \xi'))$ 
  and  $n: \text{no-T-F-symb-except-toplevel } \zeta'$ 
  have  $\text{no-T-F-symb-except-toplevel } (\text{conn } c \ (\xi @ \zeta' \# \xi'))$ 
  proof
    have  $p: \text{no-T-F-symb } (\text{conn } c \ (\xi @ \zeta \# \xi'))$ 
    using  $\text{corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F}$ 
    by  $\text{blast}$ 
    have  $l: \forall \varphi \in \text{set } (\xi @ \zeta \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$ 
    using  $\text{corr wf-conn-no-T-F-symb-iff } p$  by  $\text{blast}$ 
    from  $\text{rel incl}$  have  $\zeta' \neq FT \wedge \zeta' \neq FF$ 
    apply  $(\text{induction } \zeta \ \zeta' \text{ rule: propo-rew-step.induct})$ 
    apply  $(\text{cases rule: elim-equiv.cases, auto simp add: elim-equiv.simps})$ 
    by  $(\text{metis append-is-Nil-conv list.distinct wf-conn-list(1,2) wf-conn-no-arity-change}$ 
       $\text{wf-conn-no-arity-change-helper})+$ 
    hence  $\forall \varphi \in \text{set } (\xi @ \zeta' \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$  using  $l$  by  $\text{auto}$ 
    moreover have  $c \neq CT \wedge c \neq CF$  using  $\text{corr}$  by  $\text{auto}$ 
    ultimately show  $\text{no-T-F-symb } (\text{conn } c \ (\xi @ \zeta' \# \xi'))$ 
    by  $(\text{metis corr wf-conn-no-arity-change wf-conn-no-arity-change-helper no-T-F-symb-comp})$ 
  qed
}
ultimately show  $\text{no-T-F-except-top-level } \psi$ 
using  $\text{full-propo-rew-step-inv-stay-with-inc[of elim-equiv no-T-F-symb-except-toplevel } \varphi]$ 
 $\text{assms subformula-refl}$  unfolding  $\text{no-T-F-except-top-level-def}$  by  $\text{metis}$ 
qed

```

**lemma**  $\text{propo-rew-step-ElimImp-no-T-F: propo-rew-step elim-imp } \varphi \ \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$

**proof**  $(\text{induct rule: propo-rew-step.induct})$

**case**  $(\text{global-rel } \varphi' \ \psi')$

**thus**  $\text{no-T-F } \psi'$

**using**  $\text{elim-imp.cases no-T-F-comp-not no-T-F-decomp(1,2)}$

**by**  $(\text{metis no-T-F-comp-expanded-explicit(2)})$

**next**

**case**  $(\text{propo-rew-one-step-lift } \varphi \ \varphi' \ c \ \xi \ \xi')$

**note**  $\text{rel} = \text{this(1)}$  **and**  $\text{IH} = \text{this(2)}$  **and**  $\text{corr} = \text{this(3)}$  **and**  $\text{no-T-F} = \text{this(4)}$

{

**assume**  $c: c = CNot$

**hence**  $\text{empty: } \xi = [] \ \xi' = []$  **using**  $\text{corr}$  **by**  $\text{auto}$

**hence**  $\text{no-T-F } \varphi$  **using**  $\text{no-T-F } c \ \text{no-T-F-decomp-not}$  **by**  $\text{auto}$

**hence**  $\text{no-T-F } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$  **using**  $c \ \text{empty no-T-F-comp-not IH}$  **by**  $\text{auto}$

}

**moreover** {

**assume**  $c: c \in \text{binary-connectives}$

**then obtain**  $a \ b$  **where**  $\text{ab: } \xi @ \varphi \# \xi' = [a, b]$

**using**  $\text{corr list-length2-decomp wf-conn-bin-list-length}$  **by**  $\text{metis}$

**hence**  $\varphi: \varphi = a \vee \varphi = b$

**by**  $(\text{metis append-self-conv2 wf-conn-list-decomp(4) wf-conn-unary list.discI list.sel(3)}$   
 $\text{nth-Cons-0 tl-append2})$

**have**  $\zeta: \forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{no-T-F } \zeta$  **using**  $\text{ab } c \ \text{propo-rew-one-step-lift.premis}$  **by**  $\text{auto}$



```

hence  $\varphi'$ : no-T-F  $\varphi'$ 
  using ab IH  $\varphi$  corr no-T-F no-T-F-def all-subformula-st-decomp-explicit by auto
have  $\chi$ :  $\xi @ \varphi' \# \xi' = [\varphi', b] \vee \xi @ \varphi' \# \xi' = [a, \varphi']$ 
  by (metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps(2)
    butlast-append list.distinct(1) list.sel(3))
hence  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi')$ . no-T-F  $\zeta$  using  $\zeta$   $\varphi'$  ab by fastforce
moreover
  have no-T-F (last ( $\xi @ \varphi' \# \xi'$ )) by (simp add: calculation)
  hence no-T-F-symb (conn  $c$  ( $\xi @ \varphi' \# \xi'$ ))
    by (metis  $\chi$   $\varphi' \zeta$  ab all-subformula-st-test-symb-true-phi c last.simps list.distinct(1)
      list.set-intros(1) no-T-F-bin-decomp no-T-F-def)
  ultimately have no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) using  $c$   $\chi$  by fastforce
}
moreover {
  fix  $x$ 
  assume  $c = CVar\ x \vee c = CF \vee c = CT$ 
  hence False using corr by auto
  hence no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) by auto
}
ultimately show no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) using corr wf-conn.cases by blast
qed

```

```

lemma elim-imp-inv':
  fixes  $\varphi \psi :: 'v\ propo$ 
  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 -
  {
    {
      fix  $\varphi \psi :: 'v\ propo$ 
      have  $H$ : elim-imp  $\varphi \psi \implies$  no-T-F-except-top-level  $\varphi \implies$  no-T-F-except-top-level  $\psi$ 
        by (induct  $\varphi \psi$  rule: elim-imp.induct, auto)
    } note  $H = \text{this}$ 
    fix  $\varphi \psi :: 'v\ propo$ 
    have propo-rew-step elim-imp  $\varphi \psi \implies$  no-T-F-except-top-level  $\varphi \implies$  no-T-F-except-top-level  $\psi$ 
    proof -
      assume rel: propo-rew-step elim-imp  $\varphi \psi$ 
      and no: no-T-F-except-top-level  $\varphi$ 
      {
        assume  $\varphi = FT \vee \varphi = FF$ 
        from rel this have False
        apply (induct rule: propo-rew-step.induct)
        by (cases rule: elim-imp.cases, auto simp add: wf-conn-list(1,2))
        hence no-T-F-except-top-level  $\psi$  by blast
      }
    moreover {
      assume  $\varphi \neq FT \wedge \varphi \neq FF$ 
      hence no-T-F  $\varphi$  by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
      hence no-T-F  $\psi$  using rel propo-rew-step-ElimImp-no-T-F by blast
      hence no-T-F-except-top-level  $\psi$  by (simp add: no-T-F-no-T-F-except-top-level)
    }
    ultimately show no-T-F-except-top-level  $\psi$  by metis
  }
qed

```

```

}
moreover {
  fix  $c :: 'v \text{ connective}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $\zeta \zeta' :: 'v \text{ propo}$ 
  assume  $\text{rel: propo-rew-step elim-imp } \zeta \zeta'$ 
  and  $\text{incl: } \zeta \preceq \varphi$ 
  and  $\text{corr: wf-conn } c (\xi @ \zeta \# \xi')$ 
  and  $\text{no-T-F: no-T-F-symb-except-toplevel (conn } c (\xi @ \zeta \# \xi'))$ 
  and  $n: \text{no-T-F-symb-except-toplevel } \zeta'$ 
  have  $\text{no-T-F-symb-except-toplevel (conn } c (\xi @ \zeta' \# \xi'))$ 
  proof
    have  $p: \text{no-T-F-symb (conn } c (\xi @ \zeta \# \xi'))$ 
    by ( $\text{simp add: corr no-T-F no-T-F-symb-except-toplevel-no-T-F-symb wf-conn-list(1,2)}$ )

    have  $l: \forall \varphi \in \text{set } (\xi @ \zeta \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$ 
    using  $\text{corr wf-conn-no-T-F-symb-iff } p$  by  $\text{blast}$ 
    from  $\text{rel incl}$  have  $\zeta' \neq FT \wedge \zeta' \neq FF$ 
    apply ( $\text{induction } \zeta \zeta' \text{ rule: propo-rew-step.induct}$ )
    apply ( $\text{cases rule: elim-imp.cases, auto}$ )
    using  $\text{wf-conn-list(1,2) wf-conn-no-arity-change wf-conn-no-arity-change-helper}$ 
    by ( $\text{metis append-is-Nil-conv list.distinct(1)} +$ )
    hence  $\forall \varphi \in \text{set } (\xi @ \zeta' \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$  using  $l$  by  $\text{auto}$ 
    moreover have  $c \neq CT \wedge c \neq CF$  using  $\text{corr}$  by  $\text{auto}$ 
    ultimately show  $\text{no-T-F-symb (conn } c (\xi @ \zeta' \# \xi'))$ 
    using  $\text{corr wf-conn-no-arity-change no-T-F-symb-comp}$ 
    by ( $\text{metis wf-conn-no-arity-change-helper}$ )
  qed
}
ultimately show  $\text{no-T-F-except-top-level } \psi$ 
using  $\text{full-propo-rew-step-inv-stay-with-inc[of elim-imp no-T-F-symb-except-toplevel } \varphi]$ 
assms  $\text{subformula-refl}$  unfolding  $\text{no-T-F-except-top-level-def}$  by  $\text{metis}$ 
qed

```

### 10.3 The new CNF and DNF transformation

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

**definition**  $\text{dnf-rew}' :: 'a \text{ propo} \Rightarrow 'a \text{ propo} \Rightarrow \text{bool}$  **where**  $\text{dnf-rew}' \equiv$   
 $(\text{full (propo-rew-step elimTBFULL)}) \text{ OO}$   
 $(\text{full (propo-rew-step elim-equiv)}) \text{ OO}$   
 $(\text{full (propo-rew-step elim-imp)}) \text{ OO}$   
 $(\text{full (propo-rew-step pushNeg)}) \text{ OO}$   
 $(\text{full (propo-rew-step pushConj)})$

**lemma**  $\text{dnf-rew}'\text{-consistent: preserves-un-sat dnf-rew}'$   
**by** ( $\text{simp add: dnf-rew}'\text{-def elimEquiv-lifted-consistant elim-imp-lifted-consistant}$   
 $\text{elimTBFULL-consistent preserves-un-sat-OO pushConj-consistent pushNeg-lifted-consistant}$ )

**theorem**  $\text{cnf-transformation-correction:}$   
 $\text{dnf-rew}' \varphi \varphi' \Longrightarrow \text{is-dnf } \varphi'$   
**unfolding**  $\text{dnf-rew}'\text{-def OO-def}$   
**by** ( $\text{meson and-in-or-only-conjunction-in-disj elimTBFULL-full-propo-rew-step elim-equiv-inv}'$   
 $\text{elim-imp-inv elim-imp-inv}' \text{ is-dnf-def no-equiv-full-propo-rew-step-elim-equiv}$   
 $\text{no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(1-4)}$   
 $\text{pushNeg-full-propo-rew-step pushNeg-inv(1-3)}$ )

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'*  $\equiv$   
 (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'*  
**by** (simp add: cnf-rew'-def elimEquiv-lifted-consistant elim-imp-lifted-consistant  
 elimTBFull-consistent preserves-un-sat-OO pushDisj-consistent pushNeg-lifted-consistant)

**theorem** *cnf'-transformation-correction:*  
*cnf-rew'  $\varphi$   $\varphi' \implies$  is-cnf  $\varphi'$*   
**unfolding** *cnf-rew'-def OO-def*  
**by** (meson elimTBFull-full-propo-rew-step elim-equiv-inv' elim-imp-inv elim-imp-inv' is-cnf-def  
 no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp  
 or-in-and-only-conjunction-in-disj pushDisj-full-propo-rew-step pushDisj-inv(1-4)  
 pushNeg-full-propo-rew-step pushNeg-inv(1) pushNeg-inv(2) pushNeg-inv(3))

**end**

## 11 Partial Clausal Logic

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

### 11.1 Clauses

Clauses are (finite) multisets of literals.

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

### 11.2 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]

#### 11.2.1 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* {} **unfolding** *consistent-interp-def* **by** auto

**lemma** *consistent-interp-single*[simp]:  
*consistent-interp* {L} **unfolding** *consistent-interp-def* **by** auto

**lemma** *consistent-interp-subset:*  
**assumes**  $A \subseteq B$

and *consistent-interp*  $B$   
 shows *consistent-interp*  $A$   
 using *assms* **unfolding** *consistent-interp-def* **by** *auto*

**lemma** *consistent-interp-change-insert*:  
 $a \notin A \implies -a \notin A \implies \text{consistent-interp } (\text{insert } (-a) A) \longleftrightarrow \text{consistent-interp } (\text{insert } a A)$   
**unfolding** *consistent-interp-def* **by** *fastforce*

**lemma** *consistent-interp-insert-pos[simp]*:  
 $a \notin A \implies \text{consistent-interp } (\text{insert } a A) \longleftrightarrow \text{consistent-interp } A \wedge -a \notin A$   
**unfolding** *consistent-interp-def* **by** *auto*

**lemma** *consistent-interp-insert-not-in*:  
 $\text{consistent-interp } A \implies a \notin A \implies -a \notin A \implies \text{consistent-interp } (\text{insert } a A)$   
**unfolding** *consistent-interp-def* **by** *auto*

### 11.2.2 Atoms

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

**lemma** *atms-of-multiset[simp]*:  $\text{atms-of } (\text{mset } a) = \text{atm-of } ' \text{ set } a$   
**by** (*induct*  $a$ ) *auto*

**lemma** *atms-of-m-mset-unfold*:  
 $\text{atms-of-m } (\text{mset } ' b) = (\bigcup x \in b. \text{atm-of } ' \text{ set } x)$   
**unfolding** *atms-of-m-def* **by** *simp*

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

**lemma** *atms-of-m-empty-set[simp]*:  
 $\text{atms-of-m } \{\} = \{\}$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-m-mempty[simp]*:  
 $\text{atms-of-m } \{\{\#\}\} = \{\}$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-m-mono*:  
 $A \subseteq B \implies \text{atms-of-m } A \subseteq \text{atms-of-m } B$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-m-finite[simp]*:  
 $\text{finite } \psi s \implies \text{finite } (\text{atms-of-m } \psi s)$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-m-union[simp]*:  
 $\text{atms-of-m } (\psi s \cup \chi s) = \text{atms-of-m } \psi s \cup \text{atms-of-m } \chi s$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-m-insert[simp]*:  
 $\text{atms-of-m } (\text{insert } \psi s \chi s) = \text{atms-of-m } \psi s \cup \text{atms-of-m } \chi s$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-m-plus[simp]*:

**fixes**  $C D :: 'a \text{ literal multiset}$   
**shows**  $\text{atms-of-m } \{C + D\} = \text{atms-of-m } \{C\} \cup \text{atms-of-m } \{D\}$   
**unfolding**  $\text{atms-of-m-def}$  **by**  $\text{auto}$

**lemma**  $\text{atms-of-m-singleton[simp]}: \text{atms-of-m } \{L\} = \text{atms-of } L$   
**unfolding**  $\text{atms-of-m-def}$  **by**  $\text{auto}$

**lemma**  $\text{atms-of-atms-of-m-mono[simp]}:$   
 $A \in \psi \implies \text{atms-of } A \subseteq \text{atms-of-m } \psi$   
**unfolding**  $\text{atms-of-m-def}$  **by**  $\text{fastforce}$

**lemma**  $\text{atms-of-m-single-set-mset-atms-of[simp]}:$   
 $\text{atms-of-m } (\text{single } ' \text{ set-mset } B) = \text{atms-of } B$   
**unfolding**  $\text{atms-of-m-def}$   $\text{atms-of-def}$  **by**  $\text{auto}$

**lemma**  $\text{atms-of-m-remove-incl}:$   
**shows**  $\text{atms-of-m } (\text{Set.remove } a \ \psi) \subseteq \text{atms-of-m } \psi$   
**unfolding**  $\text{atms-of-m-def}$  **by**  $\text{auto}$

**lemma**  $\text{atms-of-m-remove-subset}:$   
 $\text{atms-of-m } (\varphi - \psi) \subseteq \text{atms-of-m } \varphi$   
**unfolding**  $\text{atms-of-m-def}$  **by**  $\text{auto}$

**lemma**  $\text{finite-atms-of-m-remove-subset[simp]}:$   
 $\text{finite } (\text{atms-of-m } A) \implies \text{finite } (\text{atms-of-m } (A - C))$   
**using**  $\text{atms-of-m-remove-subset[of } A \ C]$   $\text{finite-subset}$  **by**  $\text{blast}$

**lemma**  $\text{atms-of-m-empty-iff}:$   
 $\text{atms-of-m } A = \{\} \longleftrightarrow A = \{\{\#\}\} \vee A = \{\}$   
**apply**  $(\text{rule iffI})$   
**apply**  $(\text{metis } (\text{no-types, lifting}) \text{atms-empty-iff-empty } \text{atms-of-atms-of-m-mono } \text{insert-absorb}$   
 $\text{singleton-iff singleton-insert-inj-eq' subsetI subset-empty})$   
**apply**  $\text{auto}[]$   
**done**

**lemma**  $\text{in-implies-atm-of-on-atms-of-m}:$   
**assumes**  $L \in \# \ C \text{ and } C \in N$   
**shows**  $\text{atm-of } L \in \text{atms-of-m } N$   
**using**  $\text{atms-of-atms-of-m-mono[of } C \ N]$   $\text{assms}$  **by**  $(\text{simp add: atm-of-lit-in-atms-of subset-iff})$

**lemma**  $\text{in-plus-implies-atm-of-on-atms-of-m}:$   
**assumes**  $C + \{\#L\# \} \in N$   
**shows**  $\text{atm-of } L \in \text{atms-of-m } N$   
**using**  $\text{in-implies-atm-of-on-atms-of-m[of } C + \{\#L\# \}]$   $\text{assms}$  **by**  $\text{auto}$

**lemma**  $\text{in-m-in-literals}:$   
**assumes**  $\{\#A\# \} + D \in \psi$   
**shows**  $\text{atm-of } A \in \text{atms-of-m } \psi$   
**using**  $\text{assms}$  **by**  $(\text{auto dest: atms-of-atms-of-m-mono})$

**lemma**  $\text{atms-of-s-union[simp]}:$   
 $\text{atms-of-s } (Ia \cup Ib) = \text{atms-of-s } Ia \cup \text{atms-of-s } Ib$   
**unfolding**  $\text{atms-of-s-def}$  **by**  $\text{auto}$

**lemma** *atms-of-s-single[simp]*:  
 $atms-of-s \{L\} = \{atm-of L\}$   
**unfolding** *atms-of-s-def* **by** *auto*

**lemma** *atms-of-s-insert[simp]*:  
 $atms-of-s (insert L Ib) = \{atm-of L\} \cup atms-of-s Ib$   
**unfolding** *atms-of-s-def* **by** *auto*

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

**proof**

**assume**  $?P$

**then show**  $?Q$  **unfolding** *atms-of-s-def* **by** (*metis image-iff literal.exhaust-sel*)

**next**

**assume**  $?Q$

**then show**  $?P$  **unfolding** *atms-of-s-def* **by** *force*

**qed**

**lemma** *atm-of-in-atm-of-set-in-uminus*:  
 $atm-of L' \in atm-of 'B \implies L' \in B \vee - L' \in B$   
**using** *atms-of-s-def* **by** (*cases L'*) *fastforce+*

### 11.2.3 Totality

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

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

**lemma** *total-over-set-empty[simp]*:  
 $total-over-set I \{\}$   
**unfolding** *total-over-set-def* **by** *auto*

**lemma** *total-over-m-empty[simp]*:  
 $total-over-m I \{\}$   
**unfolding** *total-over-m-def* **by** *auto*

**lemma** *total-over-set-single[iff]*:  
 $total-over-set I \{L\} \longleftrightarrow (Pos L \in I \vee Neg L \in I)$   
**unfolding** *total-over-set-def* **by** *auto*

**lemma** *total-over-set-insert[iff]*:  
 $total-over-set I (insert L Ls) \longleftrightarrow ((Pos L \in I \vee Neg L \in I) \wedge total-over-set I Ls)$   
**unfolding** *total-over-set-def* **by** *auto*

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

**lemma** *total-over-m-subset*:  
 $A \subseteq B \implies total-over-m I B \implies total-over-m I A$   
**using** *atms-of-m-mono[of A]* **unfolding** *total-over-m-def* *total-over-set-def* **by** *auto*

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

```

using assms unfolding total-over-m-def total-over-set-def by auto

lemma total-over-m-union[iff]:
  total-over-m I (A ∪ B) ⟷ (total-over-m I A ∧ total-over-m I B)
  unfolding total-over-m-def total-over-set-def by auto

lemma total-over-m-insert[iff]:
  total-over-m I (insert a A) ⟷ (total-over-set I (atms-of a) ∧ total-over-m I A)
  unfolding total-over-m-def total-over-set-def by fastforce

lemma total-over-m-extension:
  fixes I :: 'v literal set and A :: 'v clauses
  assumes total: total-over-m I A
  shows  $\exists I'. \text{total-over-m } (I \cup I') (A \cup B)$ 
     $\wedge (\forall x \in I'. \text{atm-of } x \in \text{atms-of-m } B \wedge \text{atm-of } x \notin \text{atms-of-m } A)$ 
proof –
  let  $?I' = \{ \text{Pos } v \mid v. v \in \text{atms-of-m } B \wedge v \notin \text{atms-of-m } A \}$ 
  have  $(\forall x \in ?I'. \text{atm-of } x \in \text{atms-of-m } B \wedge \text{atm-of } x \notin \text{atms-of-m } A)$  by auto
  moreover have total-over-m (I ∪ ?I') (A ∪ B)
    using total unfolding total-over-m-def total-over-set-def by auto
  ultimately show ?thesis by blast
qed

lemma total-over-m-consistent-extension:
  fixes I :: 'v literal set and A :: 'v clauses
  assumes total: total-over-m I A
  and cons: consistent-interp I
  shows  $\exists I'. \text{total-over-m } (I \cup I') (A \cup B)$ 
     $\wedge (\forall x \in I'. \text{atm-of } x \in \text{atms-of-m } B \wedge \text{atm-of } x \notin \text{atms-of-m } A) \wedge \text{consistent-interp } (I \cup I')$ 
proof –
  let  $?I' = \{ \text{Pos } v \mid v. v \in \text{atms-of-m } B \wedge v \notin \text{atms-of-m } A \wedge \text{Pos } v \notin I \wedge \text{Neg } v \notin I \}$ 
  have  $(\forall x \in ?I'. \text{atm-of } x \in \text{atms-of-m } B \wedge \text{atm-of } x \notin \text{atms-of-m } A)$  by auto
  moreover have total-over-m (I ∪ ?I') (A ∪ B)
    using total unfolding total-over-m-def total-over-set-def by auto
  moreover have consistent-interp (I ∪ ?I')
    using cons unfolding consistent-interp-def by  $(\text{intro allI}) (\text{case-tac } L, \text{auto})$ 
  ultimately show ?thesis by blast
qed

lemma total-over-set-atms-of[simp]:
  total-over-set Ia (atms-of-s Ia)
  unfolding total-over-set-def atms-of-s-def by  $(\text{metis image-iff literal.exhaust-sel})$ 

lemma total-over-set-literal-defined:
  assumes  $\{ \#A\# \} + D \in \psi$ 
  and total-over-set I (atms-of-m ψ)
  shows  $A \in I \vee -A \in I$ 
  using assms unfolding total-over-set-def by  $(\text{metis (no-types) Neg-atm-of-iff in-m-in-literals literal.collapse(1) uminus-Neg uminus-Pos})$ 

lemma tot-over-m-remove:
  assumes total-over-m (I ∪ {L}) {ψ}
  and  $L: \neg L \in \# \psi - L \notin \# \psi$ 
  shows total-over-m I {ψ}
  unfolding total-over-m-def total-over-set-def

```

```

proof
  fix  $l$ 
  assume  $l \in \text{atms-of-}m \ \{\psi\}$ 
  then have  $\text{Pos } l \in I \vee \text{Neg } l \in I \vee l = \text{atm-of } L$ 
    using assms unfolding total-over-m-def total-over-set-def by auto
  moreover have  $\text{atm-of } L \notin \text{atms-of-}m \ \{\psi\}$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have  $\text{atm-of } L \in \text{atms-of } \psi$  by auto
    then have  $\text{Pos } (\text{atm-of } L) \in\# \ \psi \vee \text{Neg } (\text{atm-of } L) \in\# \ \psi$ 
      using atm-imp-pos-or-neg-lit by metis
    then have  $L \in\# \ \psi \vee - L \in\# \ \psi$  by (case-tac L) auto
    then show False using  $L$  by auto
  qed
  ultimately show  $\text{Pos } l \in I \vee \text{Neg } l \in I$  using  $l$  by metis
qed

```

```

lemma total-union:
  assumes total-over-m I ψ
  shows total-over-m (I ∪ I') ψ
  using assms unfolding total-over-m-def total-over-set-def by auto

```

```

lemma total-union-2:
  assumes total-over-m I ψ
  and total-over-m I' ψ'
  shows total-over-m (I ∪ I') (ψ ∪ ψ')
  using assms unfolding total-over-m-def total-over-set-def by auto

```

#### 11.2.4 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]:  $\neg I \models \{\#\}$ 
  unfolding true-cls-def by auto

```

```

lemma true-cls-singleton[iff]:  $I \models \{\#L\# \} \longleftrightarrow I \models_l L$ 
  unfolding true-cls-def by (auto split:split-if-asm)

```

```

lemma true-cls-union[iff]:  $I \models C + D \longleftrightarrow I \models C \vee I \models D$ 
  unfolding true-cls-def by auto

```

```

lemma true-cls-mono-set-mset:  $\text{set-mset } C \subseteq \text{set-mset } D \Longrightarrow I \models C \Longrightarrow I \models D$ 
  unfolding true-cls-def subset-eq Bex-mset-def by (metis mem-set-mset-iff)

```

```

lemma true-cls-mono-leD[dest]:  $A \subseteq\# \ B \Longrightarrow I \models A \Longrightarrow I \models B$ 
  unfolding true-cls-def by auto

```

```

lemma
  assumes  $I \models \psi$ 
  shows true-cls-union-increase[simp]:  $I \cup I' \models \psi$ 
  and true-cls-union-increase'[simp]:  $I' \cup I \models \psi$ 
  using assms unfolding true-cls-def by auto

```

```

lemma true-cls-mono-set-mset-l:
  assumes  $A \models \psi$ 

```



**and**  $A \subseteq B$   
**shows**  $B \models \psi$   
**using** *assms* **unfolding** *true-cls-def* **by** *auto*

**lemma** *true-cls-replicate-mset*[*iff*]:  $I \models \text{replicate-mset } n \ L \longleftrightarrow n \neq 0 \wedge I \models L$   
**by** (*induct n*) *auto*

**lemma** *true-cls-empty-entails*[*iff*]:  $\neg \{\} \models N$   
**by** (*auto simp add: true-cls-def*)

**lemma** *true-cls-not-in-remove*:  
**assumes**  $L \notin \chi$   
**and**  $I \cup \{L\} \models \chi$   
**shows**  $I \models \chi$   
**using** *assms* **unfolding** *true-cls-def* **by** *auto*

**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*]:  $I \models_s \{\}$   
**unfolding** *true-clss-def* **by** *blast*

**lemma** *true-clss-singleton*[*iff*]:  $I \models_s \{C\} \longleftrightarrow I \models C$   
**unfolding** *true-clss-def* **by** *blast*

**lemma** *true-clss-empty-entails-empty*[*iff*]:  $\{\} \models_s N \longleftrightarrow N = \{\}$   
**unfolding** *true-clss-def* **by** (*auto simp add: true-cls-def*)

**lemma** *true-cls-insert-l* [*simp*]:  
 $M \models A \implies \text{insert } L \ M \models A$   
**unfolding** *true-cls-def* **by** *auto*

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

**lemma** *true-clss-insert*[*iff*]:  $I \models_s \text{insert } C \ DD \longleftrightarrow I \models C \wedge I \models_s DD$   
**unfolding** *true-clss-def* **by** *blast*

**lemma** *true-clss-mono*:  $DD \subseteq CC \implies I \models_s CC \implies I \models_s DD$   
**unfolding** *true-clss-def* **by** *blast*

**lemma** *true-clss-union-increase*[*simp*]:  
**assumes**  $I \models_s \psi$   
**shows**  $I \cup I' \models_s \psi$   
**using** *assms* **unfolding** *true-clss-def* **by** *auto*

**lemma** *true-clss-union-increase'*[*simp*]:  
**assumes**  $I' \models_s \psi$   
**shows**  $I \cup I' \models_s \psi$   
**using** *assms* **by** (*auto simp add: true-clss-def*)

**lemma** *true-clss-commute-l*:  
 $(I \cup I' \models_s \psi) \longleftrightarrow (I' \cup I \models_s \psi)$   
**by** (*simp add: Un-commute*)

**lemma** *model-remove[simp]*:  $I \models_s N \implies I \models_s \text{Set.remove } a \ N$   
**by** (*simp add: true-clss-def*)

**lemma** *model-remove-minus[simp]*:  $I \models_s N \implies I \models_s N - A$   
**by** (*simp add: true-clss-def*)

**lemma** *notin-vars-union-true-cls-true-cls*:  
**assumes**  $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-m } A$   
**and**  $\text{atms-of } L \subseteq \text{atms-of-m } A$   
**and**  $I \cup I' \models L$   
**shows**  $I \models L$   
**using** *assms unfolding true-cls-def true-lit-def Bex-mset-def*  
**by** (*metis Un-iff atm-of-lit-in-atms-of contra-subsetD*)

**lemma** *notin-vars-union-true-clss-true-clss*:  
**assumes**  $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-m } A$   
**and**  $\text{atms-of-m } L \subseteq \text{atms-of-m } A$   
**and**  $I \cup I' \models_s L$   
**shows**  $I \models_s L$   
**using** *assms unfolding true-clss-def true-lit-def Ball-def*  
**by** (*meson atms-of-atms-of-m-mono notin-vars-union-true-cls-true-cls subset-trans*)

### 11.2.5 Satisfiability

**definition** *satisfiable* :: 'a clause set  $\Rightarrow$  bool **where**  
*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]*:  
*satisfiable*  $\{\{\#L\#\}\}$   
**unfolding** *satisfiable-def* **by** *fastforce*

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

**lemma** *satisfiable-decreasing*:  
**assumes** *satisfiable*  $(\psi \cup \psi')$   
**shows** *satisfiable*  $\psi$   
**using** *assms total-over-m-union unfolding satisfiable-def* **by** *blast*

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

**proof**  
**assume** *?B* **then show** *?sat* **by** (*auto simp add: satisfiable-def*)

**next**

**assume** *?sat*  
**then obtain**  $I$  **where**  
 $I\text{-}CC: I \models_s CC$  **and**  
 $\text{cons}: \text{consistent-interp } I$  **and**  
 $\text{tot}: \text{total-over-m } I \ CC$   
**unfolding** *satisfiable-def* **by** *auto*  
**let**  $?I = \{P. P \in I \wedge \text{atm-of } P \in \text{atms-of-m } CC\}$

**have**  $I\text{-}CC: ?I \models_s CC$   
**using**  $I\text{-}CC$  **unfolding** *true-clss-def Ball-def true-cls-def Bex-mset-def true-lit-def*

by (smt atm-of-lit-in-atms-of atms-of-atms-of-m-mono mem-Collect-eq subset-eq)  
 moreover have cons: consistent-interp ?I  
 using cons unfolding consistent-interp-def by auto  
 moreover have total-over-m ?I CC  
 using tot unfolding total-over-m-def total-over-set-def by auto  
 moreover  
 have atms-CC-incl: atms-of-m CC  $\subseteq$  atm-of'I  
 using tot unfolding total-over-m-def total-over-set-def atms-of-m-def  
 by (auto simp add: atms-of-def atms-of-s-def[symmetric])  
 have atm-of ' ?I = atms-of-m CC  
 using atms-CC-incl unfolding atms-of-m-def by force  
 ultimately show ?B by auto  
 qed

### 11.2.6 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]:  $I \models_m \{\#\}$   
 unfolding true-cls-mset-def by auto

**lemma** true-cls-mset-singleton[iff]:  $I \models_m \{\#C\} \longleftrightarrow I \models C$   
 unfolding true-cls-mset-def by (auto split: split-if-asm)

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

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

**lemma** true-cls-mset-mono:  $\text{set-mset } DD \subseteq \text{set-mset } CC \Longrightarrow I \models_m CC \Longrightarrow I \models_m DD$   
 unfolding true-cls-mset-def subset-iff by auto

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

**lemma** true-cls-mset-increasing-r[simp]:  
 $I \models_m CC \Longrightarrow I \cup J \models_m CC$   
 unfolding true-cls-mset-def by auto

**theorem** true-cls-remove-unused:  
 assumes  $I \models \psi$   
 shows  $\{v \in I. \text{atm-of } v \in \text{atms-of } \psi\} \models \psi$   
 using assms unfolding true-cls-def atms-of-def by auto

**theorem** true-clss-remove-unused:  
 assumes  $I \models_s \psi$   
 shows  $\{v \in I. \text{atm-of } v \in \text{atms-of-m } \psi\} \models_s \psi$   
 unfolding true-clss-def atms-of-def Ball-def  
**proof** (intro allI impI)  
 fix  $x$   
 assume  $x \in \psi$   
 then have  $I \models x$   
 using assms unfolding true-clss-def atms-of-def Ball-def by auto

**then have**  $\{v \in I. \text{atm-of } v \in \text{atms-of } x\} \models x$   
**by** (*simp only: true-cls-remove-unused[of I]*)  
**moreover have**  $\{v \in I. \text{atm-of } v \in \text{atms-of } x\} \subseteq \{v \in I. \text{atm-of } v \in \text{atms-of-m } \psi\}$   
**using**  $\langle x \in \psi \rangle$  **by** (*auto simp add: atms-of-m-def*)  
**ultimately show**  $\{v \in I. \text{atm-of } v \in \text{atms-of-m } \psi\} \models x$   
**using** *true-cls-mono-set-mset-l* **by** *blast*  
**qed**

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**  $I \models \psi$   
**proof** –  
**let**  $?I = \{v \in I \cup I'. \text{atm-of } v \in \text{atms-of } \psi\}$   
**have**  $?I \models \psi$  **using** *true-cls-remove-unused II'* **by** *blast*  
**moreover have**  $?I \subseteq I$  **using**  $H$  **by** *auto*  
**ultimately show** *?thesis* **using** *true-cls-mono-set-mset-l* **by** *blast*  
**qed**

**lemma** *multiset-not-empty*:  
**assumes**  $M \neq \{\#\}$   
**and**  $x \in\# M$   
**shows**  $\exists A. x = \text{Pos } A \vee x = \text{Neg } A$   
**using** *assms literal.exhaust-sel* **by** *blast*

**lemma** *atms-of-m-empty*:  
**fixes**  $\psi :: 'v \text{ clauses}$   
**assumes**  $\text{atms-of-m } \psi = \{\}$   
**shows**  $\psi = \{\} \vee \psi = \{\{\#\}\}$   
**using** *assms* **by** (*auto simp add: atms-of-m-def*)

**lemma** *consistent-interp-disjoint*:  
**assumes**  $\text{consI}: \text{consistent-interp } I$   
**and**  $\text{disj}: \text{atms-of-s } A \cap \text{atms-of-s } I = \{\}$   
**and**  $\text{consA}: \text{consistent-interp } A$   
**shows**  $\text{consistent-interp } (A \cup I)$   
**proof** (*rule ccontr*)  
**assume**  $\neg ?thesis$   
**moreover have**  $\bigwedge L. \neg (L \in A \wedge \neg L \in I)$   
**using** *disj unfolding atms-of-s-def* **by** (*auto simp add: rev-image-eqI*)  
**ultimately show** *False*  
**using**  $\text{consA consI}$  **unfolding** *consistent-interp-def* **by** (*metis (full-types) Un-iff literal.exhaust-sel uminus-Neg uminus-Pos*)  
**qed**

**lemma** *total-remove-unused*:  
**assumes**  $\text{total-over-m } I \psi$   
**shows**  $\text{total-over-m } \{v \in I. \text{atm-of } v \in \text{atms-of-m } \psi\} \psi$   
**using** *assms* **unfolding** *total-over-m-def total-over-set-def*  
**by** (*metis (lifting) literal.sel(1,2) mem-Collect-eq*)

**lemma** *true-cls-remove-hd-if-notin-vars*:  
**assumes**  $\text{insert } a \ M' \models D$

and  $\text{atm-of } a \notin \text{atms-of } D$   
 shows  $M' \models D$   
 using *assms* by (auto simp add: atm-of-lit-in-atms-of true-cls-def)

**lemma** *total-over-set-atm-of*:  
 fixes  $I :: 'v \text{ interp}$  and  $K :: 'v \text{ set}$   
 shows  $\text{total-over-set } I \ K \longleftrightarrow (\forall l \in K. l \in (\text{atm-of } I))$   
 unfolding *total-over-set-def* by (metis *atms-of-s-def in-atms-of-s-decomp*)

### 11.2.7 Tautologies

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

**lemma** *tautology-Pos-Neg[intro]*:  
 assumes  $\text{Pos } p \in \# A$  and  $\text{Neg } p \in \# A$   
 shows *tautology*  $A$   
 using *assms* unfolding *tautology-def total-over-set-def true-cls-def Bex-mset-def*  
 by (meson *atm-iff-pos-or-neg-lit true-lit-def*)

**lemma** *tautology-minus[simp]*:  
 assumes  $L \in \# A$  and  $-L \in \# A$   
 shows *tautology*  $A$   
 by (metis *assms literal.exhaust tautology-Pos-Neg uminus-Neg uminus-Pos*)

**lemma** *tautology-exists-Pos-Neg*:  
 assumes *tautology*  $\psi$   
 shows  $\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi$   
**proof** (rule *ccontr*)  
 assume  $p: \neg (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$   
 let  $?I = \{-L \mid L. L \in \# \psi\}$   
 have *total-over-set*  $?I \ (\text{atms-of } \psi)$   
   unfolding *total-over-set-def* using *atm-imp-pos-or-neg-lit* by force  
 moreover have  $\neg ?I \models \psi$   
   unfolding *true-cls-def true-lit-def Bex-mset-def* apply *clarify*  
   using  $p$  by (case-tac  $L$ ) fastforce+  
 ultimately show *False* using *assms* unfolding *tautology-def* by auto  
**qed**

**lemma** *tautology-decomp*:  
 $\text{tautology } \psi \longleftrightarrow (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$   
 using *tautology-exists-Pos-Neg* by auto

**lemma** *tautology-false[simp]*:  $\neg \text{tautology } \{\#\}$   
 unfolding *tautology-def* by auto

**lemma** *tautology-add-single*:  
 $\text{tautology } (\{\#a\} + L) \longleftrightarrow \text{tautology } L \vee -a \in \# L$   
 unfolding *tautology-decomp* by (cases  $a$ ) auto

**lemma** *minus-interp-tautology*:  
 assumes  $\{-L \mid L. L \in \# \chi\} \models \chi$   
 shows *tautology*  $\chi$   
**proof** –  
 obtain  $L$  where  $L \in \# \chi \wedge -L \in \# \chi$   
   using *assms* unfolding *true-cls-def* by auto  
 then show *?thesis* using *tautology-decomp literal.exhaust uminus-Neg uminus-Pos* by metis

qed

**lemma** *remove-literal-in-model-tautology*:

**assumes**  $I \cup \{Pos\ P\} \models \varphi$   
**and**  $I \cup \{Neg\ P\} \models \varphi$   
**shows**  $I \models \varphi \vee \text{tautology } \varphi$   
**using** *assms unfolding true-cls-def by auto*

**lemma** *tautology-imp-tautology*:

**fixes**  $\chi\ \chi' :: 'v\ \text{clause}$   
**assumes**  $\forall I. \text{total-over-m } I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$  **and** *tautology*  $\chi$   
**shows** *tautology*  $\chi'$  **unfolding** *tautology-def*

**proof** (*intro allI HOL.impI*)

**fix**  $I :: 'v\ \text{literal set}$   
**assume** *totI*: *total-over-set*  $I\ (\text{atms-of } \chi')$   
**let**  $?I' = \{Pos\ v\ |\ v. v \in \text{atms-of } \chi \wedge v \notin \text{atms-of-s } I\}$   
**have** *totI'*: *total-over-m*  $(I \cup ?I')\ \{\chi\}$  **unfolding** *total-over-m-def total-over-set-def by auto*  
**then have**  $\chi: I \cup ?I' \models \chi$  **using** *assms(2) unfolding total-over-m-def tautology-def by simp*  
**then have**  $I \cup (?I' - I) \models \chi'$  **using** *assms(1) totI' by auto*  
**moreover have**  $\bigwedge L. L \in \# \chi' \implies L \notin ?I'$   
**using** *totI unfolding total-over-set-def by (auto dest: pos-lit-in-atms-of)*  
**ultimately show**  $I \models \chi'$  **unfolding** *true-cls-def by auto*

qed

### 11.2.8 Entailment for clauses and propositions

**definition** *true-cls-cls* ::  $'a\ \text{clause} \Rightarrow 'a\ \text{clause} \Rightarrow \text{bool}$  (**infix**  $\models_f$  49) **where**

$\psi \models_f \chi \iff (\forall I. \text{total-over-m } I\ (\{\psi\} \cup \{\chi\}) \longrightarrow \text{consistent-interp } I \longrightarrow I \models \psi \longrightarrow I \models \chi)$

**definition** *true-cls-clss* ::  $'a\ \text{clause} \Rightarrow 'a\ \text{clauses} \Rightarrow \text{bool}$  (**infix**  $\models_{fs}$  49) **where**

$\psi \models_{fs} \chi \iff (\forall I. \text{total-over-m } I\ (\{\psi\} \cup \chi) \longrightarrow \text{consistent-interp } I \longrightarrow I \models \psi \longrightarrow I \models_s \chi)$

**definition** *true-clss-cls* ::  $'a\ \text{clauses} \Rightarrow 'a\ \text{clause} \Rightarrow \text{bool}$  (**infix**  $\models_p$  49) **where**

$N \models_p \chi \iff (\forall I. \text{total-over-m } I\ (N \cup \{\chi\}) \longrightarrow \text{consistent-interp } I \longrightarrow I \models_s N \longrightarrow I \models \chi)$

**definition** *true-clss-clss* ::  $'a\ \text{clauses} \Rightarrow 'a\ \text{clauses} \Rightarrow \text{bool}$  (**infix**  $\models_{ps}$  49) **where**

$N \models_{ps} N' \iff (\forall I. \text{total-over-m } I\ (N \cup N') \longrightarrow \text{consistent-interp } I \longrightarrow I \models_s N \longrightarrow I \models_s N')$

**lemma** *true-cls-cls-refl[simp]*:

$A \models_f A$   
**unfolding** *true-cls-cls-def by auto*

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

$a \models_f C \implies \text{insert } a\ A \models_p C$   
**unfolding** *true-cls-cls-def true-clss-cls-def true-clss-def by fastforce*

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

$N \models_{fs} \{\}$   
**unfolding** *true-cls-clss-def by auto*

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

$\{\varphi\} \models_p \psi \iff \varphi \models_f \psi$   
**unfolding** *true-cls-cls-def true-clss-cls-def by auto*

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

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

**unfolding** *true-clss-clss-def true-clss-clss-def* **by** *auto*

**lemma** *true-clss-clss-true-clss-clss*[*iff*]:  
 $\{\chi\} \models_{ps} \psi \longleftrightarrow \chi \models_{fs} \psi$   
**unfolding** *true-clss-clss-def true-clss-clss-def* **by** *auto*

**lemma** *true-clss-clss-empty*[*simp*]:  
 $N \models_{ps} \{\}$   
**unfolding** *true-clss-clss-def* **by** *auto*

**lemma** *true-clss-clss-subset*:  
 $A \subseteq B \implies A \models_p CC \implies B \models_p CC$   
**unfolding** *true-clss-clss-def total-over-m-union* **by** (*simp add: total-over-m-subset true-clss-mono*)

**lemma** *true-clss-clss-mono-l*[*simp*]:  
 $A \models_p CC \implies A \cup B \models_p CC$   
**by** (*auto intro: true-clss-clss-subset*)

**lemma** *true-clss-clss-mono-l2*[*simp*]:  
 $B \models_p CC \implies A \cup B \models_p CC$   
**by** (*auto intro: true-clss-clss-subset*)

**lemma** *true-clss-clss-mono-r*[*simp*]:  
 $A \models_p CC \implies A \models_p CC + CC'$   
**unfolding** *true-clss-clss-def total-over-m-union total-over-m-sum* **by** *blast*

**lemma** *true-clss-clss-mono-r'*[*simp*]:  
 $A \models_p CC' \implies A \models_p CC + CC'$   
**unfolding** *true-clss-clss-def total-over-m-union total-over-m-sum* **by** *blast*

**lemma** *true-clss-clss-union-l*[*simp*]:  
 $A \models_{ps} CC \implies A \cup B \models_{ps} CC$   
**unfolding** *true-clss-clss-def total-over-m-union* **by** *fastforce*

**lemma** *true-clss-clss-union-l-r*[*simp*]:  
 $B \models_{ps} CC \implies A \cup B \models_{ps} CC$   
**unfolding** *true-clss-clss-def total-over-m-union* **by** *fastforce*

**lemma** *true-clss-clss-in*[*simp*]:  
 $CC \in A \implies A \models_p CC$   
**unfolding** *true-clss-clss-def true-clss-def total-over-m-union* **by** *fastforce*

**lemma** *true-clss-clss-insert-l*[*simp*]:  
 $A \models_p C \implies \text{insert } a \ A \models_p C$   
**unfolding** *true-clss-clss-def true-clss-def* **using** *total-over-m-union*  
**by** (*metis Un-iff insert-is-Un sup commute*)

**lemma** *true-clss-clss-insert-l*[*simp*]:  
 $A \models_{ps} C \implies \text{insert } a \ A \models_{ps} C$   
**unfolding** *true-clss-clss-def true-clss-clss-def true-clss-def* **by** *blast*

**lemma** *true-clss-clss-union-and*[*iff*]:  
 $A \models_{ps} C \cup D \longleftrightarrow (A \models_{ps} C \wedge A \models_{ps} D)$   
**proof**  
 $\{$

```

fix A C D :: 'a clauses
assume A: A  $\models_{ps}$  C  $\cup$  D
have A  $\models_{ps}$  C
  unfolding true-clss-clss-def true-clss-clss-def insert-def total-over-m-insert
  proof (intro allI impI)
    fix I
    assume totAC: total-over-m I (A  $\cup$  C)
    and cons: consistent-interp I
    and I: I  $\models_s$  A
    then have tot: total-over-m I A and tot': total-over-m I C by auto
    obtain I' where tot': total-over-m (I  $\cup$  I') (A  $\cup$  C  $\cup$  D)
    and cons': consistent-interp (I  $\cup$  I')
    and H:  $\forall x \in I'. \text{atm-of } x \in \text{atms-of-m } D \wedge \text{atm-of } x \notin \text{atms-of-m } (A \cup C)$ 
      using total-over-m-consistent-extension[OF - cons, of A  $\cup$  C] tot tot' by blast
    moreover have I  $\cup$  I'  $\models_s$  A using I by simp
    ultimately have I  $\cup$  I'  $\models_s$  C  $\cup$  D using A unfolding true-clss-clss-def by auto
    then have I  $\cup$  I'  $\models_s$  C  $\cup$  D by auto
    then show I  $\models_s$  C using notin-vars-union-true-clss-true-clss[of I'] H by auto
  qed
} note H = this
assume A  $\models_{ps}$  C  $\cup$  D
then show A  $\models_{ps}$  C  $\wedge$  A  $\models_{ps}$  D using H[of A] Un-commute[of C D] by metis
next
assume A  $\models_{ps}$  C  $\wedge$  A  $\models_{ps}$  D
then show A  $\models_{ps}$  C  $\cup$  D
  unfolding true-clss-clss-def by auto
qed

lemma true-clss-clss-insert[iff]:
  A  $\models_{ps}$  insert L Ls  $\longleftrightarrow$  (A  $\models_p$  L  $\wedge$  A  $\models_{ps}$  Ls)
  using true-clss-clss-union-and[of A {L} Ls] by auto

lemma true-clss-clss-subset:
  A  $\subseteq$  B  $\implies$  A  $\models_{ps}$  CC  $\implies$  B  $\models_{ps}$  CC
  by (metis subset-Un-eq true-clss-clss-union-l)

lemma union-trus-clss-clss[simp]: A  $\cup$  B  $\models_{ps}$  B
  unfolding true-clss-clss-def by auto

lemma true-clss-clss-remove[simp]:
  A  $\models_{ps}$  B  $\implies$  A  $\models_{ps}$  B - C
  by (metis Un-Diff-Int true-clss-clss-union-and)

lemma true-clss-clss-subsetE:
  N  $\models_{ps}$  B  $\implies$  A  $\subseteq$  B  $\implies$  N  $\models_{ps}$  A
  by (metis sup.orderE true-clss-clss-union-and)

lemma true-clss-clss-in-imp-true-clss-clss:
  assumes N  $\models_{ps}$  U
  and A  $\in$  U
  shows N  $\models_p$  A
  using assms mk-disjoint-insert by fastforce

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

```



**unfolding** *true-clss-clss-def true-clss-def* **by** *auto*

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

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

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

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

**using** *assms unfolding true-clss-clss-def* **by** *auto*

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

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

**proof** –

**assume**  $a1: A \cup C \models_{ps} D$

**assume**  $B \models_{ps} C$

**then have**  $f2: \bigwedge M. M \cup B \models_{ps} C$

**by** (*meson true-clss-clss-union-l-r*)

**have**  $\bigwedge M. C \cup (M \cup A) \models_{ps} D$

**using**  $a1$  **by** (*simp add: Un-commute sup-left-commute*)

**then show** *?thesis*

**using**  $f2$  **by** (*metis (no-types) Un-commute true-clss-clss-left-right true-clss-clss-union-and*)

**qed**

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

**unfolding** *true-clss-clss-def*

**proof** (*intro allI impI*)

**fix**  $I$

**assume** *tot*: *total-over-m*  $I (N \cup \{D + C\})$

**and** *consistent-interp*  $I$

**and**  $I \models_s N$

{

**assume**  $L: L \in I \vee -L \in I$

**then have** *total-over-m*  $I \{D + \{\#- L\# \}\}$

**using** *tot* **by** (*cases L*) *auto*

**then have**  $I \models D + \{\#- L\# \}$  **using**  $D \langle I \models_s N \rangle$  *tot* *consistent-interp I*

**unfolding** *true-clss-clss-def* **by** *auto*

**moreover**

**have** *total-over-m*  $I \{C + \{\#L\# \}\}$

**using**  $L$  *tot* **by** (*cases L*) *auto*

**then have**  $I \models C + \{\#L\# \}$

**using**  $C \langle I \models_s N \rangle$  *tot* *consistent-interp I* **unfolding** *true-clss-clss-def* **by** *auto*

**ultimately have**  $I \models D + C$  **using**  $\langle I \models_s N \rangle$  *consistent-interp-def* **by** *fastforce*

}

**moreover** {

**assume**  $L: L \notin I \wedge -L \notin I$

**let**  $?I' = I \cup \{L\}$

**have** *consistent-interp*  $?I'$  **using**  $L \langle I \models_s N \rangle$  **by** *auto*

**moreover have** *total-over-m*  $?I' \{D + \{\#- L\# \}\}$

**using** *tot* **unfolding** *total-over-m-def total-over-set-def* **by** (*auto simp add: atms-of-def*)

**moreover have** *total-over-m*  $?I' N$  **using** *tot* **using** *total-union* **by** *blast*

**moreover have**  $?I' \models_s N$  **using**  $\langle I \models_s N \rangle$  **using** *true-clss-union-increase* **by** *blast*

**ultimately have**  $?I' \models D + \{\#- L\# \}$

**using**  $D$  **unfolding** *true-clss-clss-def* **by** *blast*

**then have**  $?I' \models D$  **using**  $L$  **by** *auto*

```

moreover
  have total-over-set  $I$  (atms-of ( $D + C$ )) using tot by auto
  then have  $L \notin \# D \wedge \neg L \notin \# D$ 
    using  $L$  unfolding total-over-set-def atms-of-def by (cases  $L$ ) force+
  ultimately have  $I \models D + C$  unfolding true-cls-def by auto
}
ultimately show  $I \models D + C$  by blast
qed

```

```

lemma atms-of-union-mset[simp]:
  atms-of ( $A \# \cup B$ ) = atms-of  $A \cup$  atms-of  $B$ 
  unfolding atms-of-def by (auto simp: max-def split: split-if-asm)

```

```

lemma true-cls-union-mset[iff]:  $I \models C \# \cup D \longleftrightarrow I \models C \vee I \models D$ 
  unfolding true-cls-def by (force simp: max-def Bex-mset-def split: split-if-asm)

```

```

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

```

```

  assumes  $D: N \models_p D + \{\# - L\# \}$ 
  and  $C: N \models_p C + \{\# L\# \}$ 
  shows  $N \models_p D \# \cup C$ 
  unfolding true-clss-cls-def

```

```

proof (intro allI impI)

```

```

  fix  $I$ 
  assume tot: total-over-m  $I$  ( $N \cup \{D \# \cup C\}$ )
  and consistent-interp  $I$ 
  and  $I \models_s N$ 
  {
    assume  $L: L \in I \vee \neg L \in I$ 
    then have total-over-m  $I$   $\{D + \{\# - L\# \}\}$ 
      using tot by (cases  $L$ ) auto
    then have  $I \models D + \{\# - L\# \}$  using  $D \langle I \models_s N \rangle$  tot consistent-interp  $I$ 
      unfolding true-clss-cls-def by auto
    moreover
      have total-over-m  $I$   $\{C + \{\# L\# \}\}$ 
        using  $L$  tot by (cases  $L$ ) auto
      then have  $I \models C + \{\# L\# \}$ 
        using  $C \langle I \models_s N \rangle$  tot consistent-interp  $I$  unfolding true-clss-cls-def by auto
      ultimately have  $I \models D \# \cup C$  using consistent-interp  $I$  unfolding consistent-interp-def
        by auto
  }

```

```

moreover {
  assume  $L: L \notin I \wedge \neg L \notin I$ 
  let  $?I' = I \cup \{L\}$ 
  have consistent-interp  $?I'$  using  $L \langle$  consistent-interp  $I \rangle$  by auto
  moreover have total-over-m  $?I'$   $\{D + \{\# - L\# \}\}$ 
    using tot unfolding total-over-m-def total-over-set-def by (auto simp add: atms-of-def)
  moreover have total-over-m  $?I' N$  using tot using total-union by blast
  moreover have  $?I' \models_s N$  using  $I \models_s N$  using true-clss-union-increase by blast
  ultimately have  $?I' \models D + \{\# - L\# \}$ 
    using  $D$  unfolding true-clss-cls-def by blast
  then have  $?I' \models D$  using  $L$  by auto
  moreover
    have total-over-set  $I$  (atms-of ( $D + C$ )) using tot by auto
    then have  $L \notin \# D \wedge \neg L \notin \# D$ 

```

```

    using L unfolding total-over-set-def atms-of-def by (cases L) force+
    ultimately have  $I \models D \# \cup C$  unfolding true-cls-def by auto
  }
  ultimately show  $I \models D \# \cup C$  by blast
qed

```

```

lemma satisfiable-carac[iff]:
  ( $\exists I. \text{consistent-interp } I \wedge I \models_s \varphi$ )  $\longleftrightarrow$  satisfiable  $\varphi$  (is ( $\exists I. ?Q I$ )  $\longleftrightarrow ?S$ )
proof
  assume ?S
  then show  $\exists I. ?Q I$  unfolding satisfiable-def by auto
next
  assume  $\exists I. ?Q I$ 
  then obtain I where cons: consistent-interp I and I:  $I \models_s \varphi$  by metis
  let ?I' = {Pos v | v.  $v \notin \text{atms-of-s } I \wedge v \in \text{atms-of-m } \varphi$ }
  have consistent-interp ( $I \cup ?I'$ )
    using cons unfolding consistent-interp-def by (intro allI) (case-tac L, auto)
  moreover have total-over-m ( $I \cup ?I'$ )  $\varphi$ 
    unfolding total-over-m-def total-over-set-def by auto
  moreover have  $I \cup ?I' \models_s \varphi$ 
    using I unfolding Ball-def true-clss-def true-cls-def by auto
  ultimately show ?S unfolding satisfiable-def by blast
qed

```

```

lemma satisfiable-carac'[simp]: consistent-interp I  $\implies I \models_s \varphi \implies$  satisfiable  $\varphi$ 
  using satisfiable-carac by metis

```

### 11.3 Subsumptions

```

lemma subsumption-total-over-m:
  assumes  $A \subseteq \# B$ 
  shows total-over-m  $I \{B\} \implies$  total-over-m  $I \{A\}$ 
  using assms atms-of-m-plus unfolding subset-mset-def total-over-m-def total-over-set-def
  by (auto simp add: mset-le-exists-conv)

```

```

lemma atm-of-eq-atm-of:
  atm-of  $L = \text{atm-of } L' \longleftrightarrow (L = L' \vee L = -L')$ 
  by (cases L; cases L') auto

```

```

lemma atms-of-replicate-mset-replicate-mset-uminus[simp]:
  atms-of ( $D - \text{replicate-mset } (\text{count } D \ L) \ L - \text{replicate-mset } (\text{count } D \ (-L)) \ (-L)$ )
  = atms-of  $D - \{\text{atm-of } L\}$ 
  by (auto split: split-if-asm simp add: atm-of-eq-atm-of atms-of-def)

```

```

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  $(\forall I. \text{total-over-m } I \{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \vee$  tautology  $\varphi$ 
  using assms
proof (induct card {Pos v | v.  $v \in \text{atms-of } D \wedge v \notin \text{atms-of } C$ } arbitrary: D
  rule: nat-less-induct-case)
  case 0 note  $n = \text{this}(1)$  and  $H = \text{this}(2)$  and  $\text{incl} = \text{this}(3)$ 
  then have atms-of  $D \subseteq \text{atms-of } C$  by auto
  then have  $\forall I. \text{total-over-m } I \{C\} \longrightarrow \text{total-over-m } I \{D\}$ 
    unfolding total-over-m-def total-over-set-def by auto
  moreover have  $\forall I. I \models C \longrightarrow I \models D$  using incl true-cls-mono-leD by blast

```

```

ultimately show ?case using H by auto
next
case (Suc n D) note IH = this(1) and card = this(2) and H = this(3) and incl = this(4)
let ?atms = {Pos v | v. v ∈ atms-of D ∧ v ∉ atms-of C}
have finite ?atms by auto
then obtain L where L: L ∈ ?atms
  using card by (metis (no-types, lifting) Collect-empty-eq card-0-eq mem-Collect-eq
    nat.simps(3))
let ?D' = D - replicate-mset (count D L) L - replicate-mset (count D (-L)) (-L)
have atms-of-D: atms-of-m {D} ⊆ atms-of-m {?D'} ∪ {atm-of L} by auto

{
  fix I
  assume total-over-m I {?D'}
  then have tot: total-over-m (I ∪ {L}) {D}
    unfolding total-over-m-def total-over-set-def using atms-of-D by auto

  assume IDL: I ⊨ ?D'
  then have I ∪ {L} ⊨ D unfolding true-cls-def by force
  then have I ∪ {L} ⊨ φ using H tot by auto

  moreover
  have tot': total-over-m (I ∪ {-L}) {D}
    using tot unfolding total-over-m-def total-over-set-def by auto
  have I ∪ {-L} ⊨ D using IDL unfolding true-cls-def by force
  then have I ∪ {-L} ⊨ φ using H tot' by auto
  ultimately have I ⊨ φ ∨ tautology φ
    using L remove-literal-in-model-tautology by force
} note H' = this

have L ∉# C and -L ∉# C using L atm-iff-pos-or-neg-lit by force+
then have C-in-D': C ⊆# ?D' using ⟨C ⊆# D⟩ by (auto simp add: subseq-mset-def)
have card {Pos v | v. v ∈ atms-of ?D' ∧ v ∉ atms-of C} <
  card {Pos v | v. v ∈ atms-of D ∧ v ∉ atms-of C}
  using L by (auto intro!: psubset-card-mono)
then show ?case
  using IH C-in-D' H' unfolding card[symmetric] by blast
qed

```

## 11.4 Removing Duplicates

```

lemma tautology-remdups-mset[iff]:
  tautology (remdups-mset C) ⟷ tautology C
  unfolding tautology-decomp by auto

lemma atms-of-remdups-mset[simp]: atms-of (remdups-mset C) = atms-of C
  unfolding atms-of-def by auto

lemma true-cls-remdups-mset[iff]: I ⊨ remdups-mset C ⟷ I ⊨ C
  unfolding true-cls-def by auto

lemma true-clss-cls-remdups-mset[iff]: A ⊨p remdups-mset C ⟷ A ⊨p C
  unfolding true-clss-cls-def total-over-m-def by auto

```

## 11.5 Set of all Simple Clauses

A simple clause contains no duplicate and is not tautology.

**function** *build-all-simple-clss* :: 'v :: linorder set  $\Rightarrow$  'v clause set **where**

```

build-all-simple-clss vars =
  (if  $\neg$ finite vars  $\vee$  vars = {}
   then {{#}}
   else
     let cls' = build-all-simple-clss (vars - {Min vars}) in
     {{#Pos (Min vars)#} +  $\chi$  |  $\chi$ .  $\chi \in$  cls'}  $\cup$ 
     {{#Neg (Min vars)#} +  $\chi$  |  $\chi$ .  $\chi \in$  cls'}  $\cup$ 
     cls')

```

**by** *auto*

**termination by** (*relation measure card*) (*auto simp add: card-gt-0-iff*)

To avoid infinite simplifier loops:

**declare** *build-all-simple-clss.simps*[*simp del*]

**lemma** *build-all-simple-clss-simps-if*[*simp*]:

```

 $\neg$ finite vars  $\vee$  vars = {}  $\implies$  build-all-simple-clss vars = {{#}}
by (simp add: build-all-simple-clss.simps)

```

**lemma** *build-all-simple-clss-simps-else*[*simp*]:

**fixes** vars :: 'v :: linorder set

**defines** cls  $\equiv$  *build-all-simple-clss* (vars - {Min vars})

**shows**

```

finite vars  $\wedge$  vars  $\neq$  {}  $\implies$  build-all-simple-clss (vars :: 'v :: linorder set) =
  {{#Pos (Min vars)#} +  $\chi$  |  $\chi$ .  $\chi \in$  cls}
 $\cup$  {{#Neg (Min vars)#} +  $\chi$  |  $\chi$ .  $\chi \in$  cls}
 $\cup$  cls

```

**using** *build-all-simple-clss.simps*[*of vars*] **unfolding** *Let-def cls-def* **by** *metis*

**lemma** *build-all-simple-clss-finite*:

**fixes** atms :: 'v :: linorder set

**shows** finite (*build-all-simple-clss* atms)

**proof** (*induct card atms arbitrary: atms rule: nat-less-induct*)

**case** (1 atms) **note** *IH* = *this*

```

{
  assume atms = {}  $\vee$   $\neg$ finite atms
  then have finite (build-all-simple-clss atms) by auto
}

```

**moreover** {

assume atms: atms  $\neq$  {} **and** fin: finite atms

then have Min atms  $\in$  atms **using** *Min-in* **by** *auto*

then have card (atms - {Min atms}) < card atms **using** fin atms **by** (*meson card-Diff1-less*)

then have finite (*build-all-simple-clss* (atms - {Min atms})) **using** *IH* **by** *auto*

then have finite (*build-all-simple-clss* atms) **by** (*simp add: atms fin*)

}

**ultimately show** finite (*build-all-simple-clss* atms) **by** *blast*

**qed**

**lemma** *build-all-simple-clssE*:

**assumes**

$x \in$  *build-all-simple-clss* atms **and**

finite atms

```

shows  $\text{atms-of } x \subseteq \text{atms} \wedge \neg \text{tautology } x \wedge \text{distinct-mset } x$ 
using assms
proof (induct card atms arbitrary: atms x)
  case ( $0 \text{ atms}$ )
  then show ?case by auto
next
  case ( $\text{Suc } n$ ) note  $\text{IH} = \text{this}(1)$  and  $\text{card} = \text{this}(2)$  and  $x = \text{this}(3)$  and  $\text{finite} = \text{this}(4)$ 
  obtain  $v$  where  $v \in \text{atms}$  and  $v: v = \text{Min } \text{atms}$ 
  using Min-in card local.finite by fastforce

  let  $\text{?atms}' = \text{atms} - \{v\}$ 
  have build-all-simple-clss atms
    =  $\{\{\# \text{Pos } v\# \} + \chi \mid \chi. \chi \in \text{build-all-simple-clss } (\text{?atms}')\}$ 
     $\cup \{\{\# \text{Neg } v\# \} + \chi \mid \chi. \chi \in \text{build-all-simple-clss } (\text{?atms}')\}$ 
     $\cup \text{build-all-simple-clss } (\text{?atms}')$ 
  using build-all-simple-clss-simps-else[of atms] finite  $\langle v \in \text{atms} \rangle$  unfolding  $v$ 
  by (metis emptyE)
  then consider
    ( $\text{Pos}$ )  $\chi \varphi$  where  $x = \{\# \varphi\# \} + \chi$  and  $\chi \in \text{build-all-simple-clss } (\text{?atms}')$  and
     $\varphi = \text{Pos } v \vee \varphi = \text{Neg } v$ 
    | ( $\text{In}$ )  $x \in \text{build-all-simple-clss } (\text{?atms}')$ 
  using  $x$  by auto
  then show ?case
  proof cases
    case  $\text{In}$ 
    then show ?thesis using card finite IH[of ?atms']  $\langle v \in \text{atms} \rangle$  by fastforce
  next
  case  $\text{Pos}$  note  $x\text{-}\chi = \text{this}(1)$  and  $\chi = \text{this}(2)$  and  $\varphi = \text{this}(3)$ 
  have
     $\text{atms-of } \chi \subseteq \text{atms} - \{v\}$  and
     $\neg \text{tautology } \chi$  and
     $\text{distinct-mset } \chi$ 
    using card finite IH[of ?atms' ]  $\langle v \in \text{atms} \rangle$   $x\text{-}\chi$   $\chi$  by auto
  moreover then have  $\text{count } \chi (\text{Neg } v) = 0$ 
    using  $\langle v \in \text{atms} \rangle$  unfolding  $x\text{-}\chi$  by (metis Diff-insert-absorb Set.set-insert
    atm-iff-pos-or-neg-lit gr0I subset-iff)
  moreover have  $\text{count } \chi (\text{Pos } v) = 0$ 
    using  $\langle \text{atms-of } \chi \subseteq \text{atms} - \{v\} \rangle$  by (meson Diff-iff atm-iff-pos-or-neg-lit
    contra-subsetD insertI1 not-gr0)
  ultimately show ?thesis
    using  $\langle v \in \text{atms} \rangle \varphi$  unfolding  $x\text{-}\chi$ 
    by (auto simp add: tautology-add-single distinct-mset-add-single)
  qed
qed

lemma cls-in-build-all-simple-clss:
  shows  $\{\#\} \in \text{build-all-simple-clss } s$ 
  by (induct s rule: build-all-simple-clss.induct)
  (metis (no-types, lifting) UnCI build-all-simple-clss.simps insertI1)

lemma build-all-simple-clss-card:
  fixes  $\text{atms} :: 'v :: \text{linorder set}$ 
  assumes finite atms
  shows  $\text{card } (\text{build-all-simple-clss } \text{atms}) \leq 3 \wedge (\text{card } \text{atms})$ 
  using assms

```

**proof** (*induct card atms arbitrary: atms rule: nat-less-induct*)  
**case** (*1 atms*) **note** *IH = this(1)* **and** *finite = this(2)*  
{  
  **assume** *atms = {}*  
  **then have** *card (build-all-simple-clss atms) ≤ 3 ^ (card atms)* **by** *auto*  
}  
**moreover** {  
  **let** *?P = {{#Pos (Min atms)#} + χ | χ. χ ∈ build-all-simple-clss (atms - {Min atms})}*  
  **let** *?N = {{#Neg (Min atms)#} + χ | χ. χ ∈ build-all-simple-clss (atms - {Min atms})}*  
  **let** *?Z = build-all-simple-clss (atms - {Min atms})*  
  **assume** *atms: atms ≠ {}*  
  **then have** *min: Min atms ∈ atms* **using** *Min-in finite* **by** *auto*  
  **then have** *card-atms-1: card atms ≥ 1* **by** (*simp add: Suc-leI atms card-gt-0-iff local.finite*)  
  **have** *card (build-all-simple-clss atms) = card (?P ∪ ?N ∪ ?Z)* **using** *atms finite* **by** *simp*  
  **moreover**  
    **have**  $\bigwedge M Ma. \text{card } ((M::'v \text{ literal multiset set}) \cup Ma) \leq \text{card } Ma + \text{card } M$   
    **by** (*simp add: add commute card-Un-le*)  
    **then have** *card (?P ∪ ?N ∪ ?Z) ≤ card ?Z + (card ?P + card ?N)*  
    **by** (*meson Nat.le-trans card-Un-le nat-add-left-cancel-le*)  
    **then have** *card (?P ∪ ?N ∪ ?Z) ≤ card ?P + card ?N + card ?Z*  
  
    **by** *presburger*  
  **also**  
    **have** *PZ: card ?P ≤ card ?Z*  
    **by** (*simp add: Setcompr-eq-image build-all-simple-clss-finite card-image-le*)  
    **have** *NZ: card ?N ≤ card ?Z*  
    **by** (*simp add: Setcompr-eq-image build-all-simple-clss-finite card-image-le*)  
    **have** *card ?P + card ?N + card ?Z ≤ card ?Z + card ?Z + card ?Z*  
    **using** *PZ NZ* **by** *linarith*  
  **finally have** *card (build-all-simple-clss atms) ≤ card ?Z + card ?Z + card ?Z .*  
  **moreover**  
    **have** *finite': finite (atms - {Min atms}) and*  
    *card: card (atms - {Min atms}) = card atms - 1*  
    **using** *finite min* **by** *auto*  
    **have** *card-inf: card (atms - {Min atms}) < card atms*  
    **using** *card (card atms ≥ 1) min* **by** *auto*  
    **then have** *card ?Z ≤ 3 ^ (card atms - 1)* **using** *IH finite' card* **by** *metis*  
  **moreover**  
    **have**  $(3::nat) ^ (\text{card atms} - 1) + 3 ^ (\text{card atms} - 1) + 3 ^ (\text{card atms} - 1)$   
     $= 3 * 3 ^ (\text{card atms} - 1)$  **by** *simp*  
    **then have**  $(3::nat) ^ (\text{card atms} - 1) + 3 ^ (\text{card atms} - 1) + 3 ^ (\text{card atms} - 1)$   
     $= 3 ^ (\text{card atms})$  **by** (*metis card card-Suc-Diff1 local.finite min power-Suc*)  
  **ultimately have** *card (build-all-simple-clss atms) ≤ 3 ^ (card atms)* **by** *linarith*  
}  
**ultimately show** *card (build-all-simple-clss atms) ≤ 3 ^ (card atms)* **by** *metis*  
**qed**

**lemma** *build-all-simple-clss-mono-disj:*

**assumes** *atms ∩ atms' = {}* **and** *finite atms* **and** *finite atms'*  
**shows** *build-all-simple-clss atms ⊆ build-all-simple-clss (atms ∪ atms')*  
**using** *assms*  
**proof** (*induct card (atms ∪ atms') arbitrary: atms atms'*)  
**case** (*0 atms' atms*)  
**then show** *?case* **by** *auto*  
**next**

```

case (Suc n atms atms') note IH = this(1) and c = this(2) and disj = this(3) and finite = this(4)
and finite' = this(5)
let ?min = Min (atms ∪ atms')
have m: ?min ∈ atms ∨ ?min ∈ atms' by (metis Min-in Un-iff c card-eq-0-iff nat.distinct(1))
moreover {
  assume min: ?min ∈ atms'
  then have min': ?min ∉ atms using disj by auto
  then have atms = atms - {?min} by fastforce
  then have n = card (atms ∪ (atms' - {?min}))
    using c min finite finite' by (metis Min-in Un-Diff card-Diff-singleton-if diff-Suc-1
      finite-UnI sup-eq-bot-iff)
  moreover have atms ∩ (atms' - {?min}) = {} using disj by auto
  moreover have finite (atms' - {?min}) using finite' by auto
  ultimately have build-all-simple-clss atms ⊆ build-all-simple-clss (atms ∪ (atms' - {?min}))
    using IH[of atms atms' - {?min}] finite by metis
  moreover have atms ∪ (atms' - {?min}) = (atms ∪ atms') - {?min} using min min' by auto
  ultimately have ?case by (metis (no-types, lifting) build-all-simple-clss.simps c card-0-eq
    finite' finite-UnI le-supI2 local.finite nat.distinct(1))
}
moreover {
  let ?atms' = atms - {Min atms}
  assume min: ?min ∈ atms
  moreover have min': ?min ∉ atms' using disj min by auto
  moreover have atms' - {?min} = atms'
    using ⟨?min ∉ atms'⟩ by fastforce
  ultimately have n = card (atms - {?min} ∪ atms')
    by (metis Min-in Un-Diff c card-0-eq card-Diff-singleton-if diff-Suc-1 finite' finite-Un
      finite nat.distinct(1))
  moreover have finite (atms - {?min}) using finite by auto
  moreover have (atms - {?min}) ∩ atms' = {} using disj by auto
  ultimately have build-all-simple-clss (atms - {?min})
    ⊆ build-all-simple-clss ((atms - {?min}) ∪ atms')
    using IH[of atms - {?min} atms'] finite' by metis
  moreover have build-all-simple-clss atms
    = {{#Pos (Min atms)#} + χ | χ. χ ∈ build-all-simple-clss (?atms')}
      ∪ {{#Neg (Min atms)#} + χ | χ. χ ∈ build-all-simple-clss (?atms')}
      ∪ build-all-simple-clss (?atms')
    using build-all-simple-clss-simps-else[of atms] finite min by (metis emptyE)
  moreover
    let ?mcls = build-all-simple-clss (atms ∪ atms' - {?min})
    have build-all-simple-clss (atms ∪ atms')
      = {{#Pos (?min)#} + χ | χ. χ ∈ ?mcls} ∪ {{#Neg (?min)#} + χ | χ. χ ∈ ?mcls} ∪ ?mcls
    using build-all-simple-clss-simps-else[of atms ∪ atms'] finite' min
    by (metis c card-eq-0-iff nat.distinct(1))
  moreover have atms ∪ atms' - {?min} = atms - {?min} ∪ atms'
    using min min' by (simp add: Un-Diff)
  moreover have Min atms = ?min using min min' by (simp add: Min-eqI finite' local.finite)
  ultimately have ?case by auto
}
ultimately show ?case by metis
qed

```

**lemma** build-all-simple-clss-mono:

**assumes** finite: finite atms' **and** incl: atms ⊆ atms'  
**shows** build-all-simple-clss atms ⊆ build-all-simple-clss atms'



**proof** –

**have**  $atms' = atms \cup (atms' - atms)$  **using** *incl* **by** *auto*  
**moreover have**  $finite (atms' - atms)$  **using** *finite* **by** *auto*  
**moreover have**  $atms \cap (atms' - atms) = \{\}$  **by** *auto*  
**ultimately show** *?thesis*  
**using** *rev-finite-subset[OF assms]* *build-all-simple-clss-mono-disj* **by** (*metis (no-types)*)  
**qed**

**lemma** *distinct-mset-not-tautology-implies-in-build-all-simple-clss:*

**assumes** *distinct-mset*  $\chi$  **and**  $\neg tautology \chi$   
**shows**  $\chi \in build-all-simple-clss (atms-of \chi)$   
**using** *assms*

**proof** (*induct card (atms-of  $\chi$ ) arbitrary:  $\chi$* )

**case** 0

**then show** *?case* **by** *simp*

**next**

**case** (*Suc n*) **note**  $IH = this(1)$  **and**  $simp = this(3)$  **and**  $c = this(2)$  **and**  $no-dup = this(4)$   
**have** *finite: finite (atms-of  $\chi$ )* **by** *simp*

**with** *no-dup atm-iff-pos-or-neg-lit* **obtain**  $L$  **where**

$L\chi: L \in\# \chi$  **and**

$L-min: atm-of L = Min (atms-of \chi)$  **and**

$mL\chi: \neg -L \in\# \chi$

**by** (*metis Min-in c card-0-eq literal.sel(1,2) nat.distinct(1) tautology-minus*)

**then have**  $\chi L: \chi = (\chi - \{\#L\# \}) + \{\#L\# \}$  **by** *auto*

**have**  $atm\chi: atms-of \chi = atms-of (\chi - \{\#L\# \}) \cup \{atm-of L\}$

**using** *arg-cong[OF  $\chi L$ , of atms-of]* **by** *simp*

**have**  $a\chi: atms-of (\chi - \{\#L\# \}) = (atms-of \chi) - \{atm-of L\}$

**proof** (*standard, standard*)

**fix**  $v$

**assume**  $a: v \in atms-of (\chi - \{\#L\# \})$

**then obtain**  $l$  **where**  $l: v = atm-of l$  **and**  $l': l \in\# \chi - \{\#L\# \}$

**unfolding** *atms-of-def* **by** *auto*

**moreover** {

**assume**  $v = atm-of L$

**then have**  $L \in\# \chi - \{\#L\# \} \vee -L \in\# \chi - \{\#L\# \}$

**using**  $l' l$  **by** (*auto simp add: atm-of-eq-atm-of*)

**moreover have**  $L \notin\# \chi - \{\#L\# \}$  **using**  $L \in\# \chi$  *simp* **unfolding** *distinct-mset-def* **by** *auto*

**ultimately have** *False* **using**  $mL\chi$  **by** *auto*

}

**ultimately show**  $v \in atms-of \chi - \{atm-of L\}$

**by** (*auto dest: atm-of-lit-in-atms-of split: split-if-asm*)

**next**

**show**  $atms-of \chi - \{atm-of L\} \subseteq atms-of (\chi - \{\#L\# \})$  **using**  $atm\chi$  **by** *auto*

**qed**

**let**  $?s' = build-all-simple-clss (atms-of (\chi - \{\#L\# \}))$

**have**  $card (atms-of (\chi - \{\#L\# \})) = n$

**using**  $c$  *finite*  $a\chi$  **by** (*simp add:  $L\chi$  atm-of-lit-in-atms-of*)

**moreover have** *distinct-mset*  $(\chi - \{\#L\# \})$  **using** *simp* **by** *auto*

**moreover have**  $\neg tautology (\chi - \{\#L\# \})$

**by** (*meson Multiset.diff-le-self mset-leD no-dup tautology-decomp*)

**ultimately have**  $\chi in: \chi - \{\#L\# \} \in build-all-simple-clss (atms-of (\chi - \{\#L\# \}))$

**using**  $IH$  **by** *simp*

```

have  $\chi = \{\#L\# \} + (\chi - \{\#L\# \})$  using  $\chi L$  by (simp add: add.commute)
then show ?case
  using  $\chi$  in  $L$ -min  $a\chi$ 
  by (cases  $L$ )
    (auto simp add: build-all-simple-cls.simps[of atms-of  $\chi$ ] Let-def)
qed

lemma simplified-in-build-all:
  assumes finite  $\psi$  and distinct-mset-set  $\psi$  and  $\forall \chi \in \psi. \neg \text{tautology } \chi$ 
  shows  $\psi \subseteq \text{build-all-simple-cls } (\text{atms-of-} m \ \psi)$ 
  using assms
proof (induct rule: finite.induct)
  case emptyI
  then show ?case by simp
next
  case (insertI  $\psi \ \chi$ ) note finite = this(1) and IH = this(2) and simp = this(3) and tauto = this(4)
  have distinct-mset  $\chi$  and  $\neg \text{tautology } \chi$ 
    using simp tauto unfolding distinct-mset-set-def by auto
  from distinct-mset-not-tautology-implies-in-build-all-simple-cls[OF this]
  have  $\chi: \chi \in \text{build-all-simple-cls } (\text{atms-of } \chi)$  .
  then have  $\psi \subseteq \text{build-all-simple-cls } (\text{atms-of-} m \ \psi)$  using IH simp tauto by auto
  moreover
    have  $\text{atms-of-} m \ \psi \subseteq \text{atms-of-} m \ (\text{insert } \chi \ \psi)$  unfolding atms-of-m-def atms-of-def by force
  ultimately
    have  $\psi \subseteq \text{build-all-simple-cls } (\text{atms-of-} m \ (\text{insert } \chi \ \psi))$ 
      by (meson atms-of-m-finite build-all-simple-cls-mono dual-order.trans finite.insertI local.finite)
  moreover
    have  $\chi \in \text{build-all-simple-cls } (\text{atms-of-} m \ (\text{insert } \chi \ \psi))$ 
      using  $\chi$  finite build-all-simple-cls-mono[of atms-of- $m \ (\text{insert } \chi \ \psi)$ ] by auto
  ultimately show ?case by auto
qed

```

## 11.6 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]:
   $I \models_{es} \{\}$ 
  unfolding entails-def by auto

```

```

lemma entails-single[iff]:
   $I \models_{es} \{a\} \longleftrightarrow I \models_e a$ 
  unfolding entails-def by auto

```

```

lemma entails-insert-l[simp]:
   $M \models_{es} A \implies \text{insert } L \ M \models_{es} A$ 
  unfolding entails-def by (metis Un-commute entail-union insert-is-Un)

```

**lemma** *entails-union*[*iff*]:  $I \models_{es} CC \cup DD \longleftrightarrow I \models_{es} CC \wedge I \models_{es} DD$   
**unfolding** *entails-def* **by** *blast*

**lemma** *entails-insert*[*iff*]:  $I \models_{es} \text{insert } C \text{ } DD \longleftrightarrow I \models_e C \wedge I \models_{es} DD$   
**unfolding** *entails-def* **by** *blast*

**lemma** *entails-insert-mono*:  $DD \subseteq CC \implies I \models_{es} CC \implies I \models_{es} DD$   
**unfolding** *entails-def* **by** *blast*

**lemma** *entails-union-increase*[*simp*]:  
**assumes**  $I \models_{es} \psi$   
**shows**  $I \cup I' \models_{es} \psi$   
**using** *assms* **unfolding** *entails-def* **by** *auto*

**lemma** *true-clss-commute-l*:  
 $(I \cup I' \models_{es} \psi) \longleftrightarrow (I' \cup I \models_{es} \psi)$   
**by** (*simp add: Un-commute*)

**lemma** *entails-remove*[*simp*]:  $I \models_{es} N \implies I \models_{es} \text{Set.remove } a \text{ } N$   
**by** (*simp add: entails-def*)

**lemma** *entails-remove-minus*[*simp*]:  $I \models_{es} N \implies I \models_{es} N - A$   
**by** (*simp add: entails-def*)

**end**

**interpretation** *true-cls*: *entail true-cls*  
**by** *standard* (*auto simp add: true-cls-def*)

## 11.7 Entailment to be extended

**definition** *true-clss-ext* :: '*a literal set*  $\Rightarrow$  '*a literal multiset set*  $\Rightarrow$  *bool* (**infix**  $\models_{sext}$  49)  
**where**  
 $I \models_{sext} N \longleftrightarrow (\forall J. I \subseteq J \longrightarrow \text{consistent-interp } J \longrightarrow \text{total-over-m } J \text{ } N \longrightarrow J \models_s N)$

**lemma** *true-clss-imp-true-cls-ext*:  
 $I \models_s N \implies I \models_{sext} N$   
**unfolding** *true-clss-ext-def* **by** (*metis sup.orderE true-clss-union-increase'*)

**lemma** *true-clss-ext-decrease-right-remove-r*:  
**assumes**  $I \models_{sext} N$   
**shows**  $I \models_{sext} N - \{C\}$   
**unfolding** *true-clss-ext-def*  
**proof** (*intro allI impI*)  
**fix**  $J$   
**assume**  
 $I \subseteq J$  **and**  
*cons*: *consistent-interp*  $J$  **and**  
*tot*: *total-over-m*  $J$   $(N - \{C\})$   
**let**  $?J = J \cup \{\text{Pos } (\text{atm-of } P) | P. P \in\# C \wedge \text{atm-of } P \notin \text{atm-of } J\}$   
**have**  $I \subseteq ?J$  **using**  $I \subseteq J$  **by** *auto*  
**moreover** **have** *consistent-interp*  $?J$   
**using** *cons* **unfolding** *consistent-interp-def* **apply**  $-$   
**apply** (*rule allI*) **by** (*case-tac L*) (*fastforce simp add: image-iff*)  
**moreover**  
**have** *ex-or-eq*:  $\bigwedge l R J. \exists P. (l = P \vee l = -P) \wedge P \in\# C \wedge P \notin J \wedge -P \notin J$

```

     $\longleftrightarrow (l \in \# C \wedge l \notin J \wedge -l \notin J) \vee (-l \in \# C \wedge l \notin J \wedge -l \notin J)$ 
    by (metis uminus-of-uminus-id)
  have total-over-m ?J N

  using tot unfolding total-over-m-def total-over-set-def atms-of-m-def
  apply (auto simp add:atms-of-def)
  apply (case-tac a  $\in N - \{C\}$ )
  apply auto[]
  using atms-of-s-def atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set by fastforce+
  ultimately have ?J  $\models_s N$ 
  using assms unfolding true-clss-ext-def by blast
  then have ?J  $\models_s N - \{C\}$  by auto
  have  $\{v \in ?J. \text{atm-of } v \in \text{atms-of-m } (N - \{C\})\} \subseteq J$ 
  by (smt UnCI  $\langle \text{consistent-interp } (J \cup \{\text{Pos } (\text{atm-of } P) \mid P. P \in \# C \wedge \text{atm-of } P \notin \text{atm-of } J \rangle$ 
    atm-of-in-atm-of-set-in-uminus consistent-interp-def mem-Collect-eq subsetI tot
    total-over-m-def total-over-set-atm-of)
  then show J  $\models_s N - \{C\}$ 
  using true-clss-remove-unused[OF  $\langle ?J \models_s N - \{C\} \rangle$ ] unfolding true-clss-def
  by (meson true-clss-mono-set-mset-l)
qed

```

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

```

  assumes consistent-interp I and I  $\models_{\text{sext}} A$ 
  shows satisfiable A
  by (metis Un-empty-left assms satisfiable-carac subset-Un-eq sup.left-idem
    total-over-m-consistent-extension total-over-m-empty true-clss-ext-def)

```

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

```

  assumes  $\neg \text{consistent-interp } I$ 
  shows I  $\not\models_{\text{sext}} A$ 
  by (meson assms consistent-interp-subset true-clss-ext-def)
end

```

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

**begin**

## 12 Resolution

### 12.1 Simplification Rules

**inductive** *simplify* :: '*v* clauses  $\Rightarrow$  '*v* clauses  $\Rightarrow$  bool **for** *N* :: '*v* clause set **where**

*tautology-deletion*:

$(A + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}) \in N \Longrightarrow \text{simplify } N (N - \{A + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\})$

*condensation*:

$(A + \{\# L\} + \{\# L\}) \in N \Longrightarrow \text{simplify } N (N - \{A + \{\# L\} + \{\# L\}\} \cup \{A + \{\# L\}\})$

*subsumption*:

$A \in N \Longrightarrow A \subset \# B \Longrightarrow B \in N \Longrightarrow \text{simplify } N (N - \{B\})$

**lemma** *simplify-preserves-un-sat'*:

```

  fixes N N' :: 'v clauses
  assumes simplify N N'
  and total-over-m I N
  shows I  $\models_s N' \longrightarrow I \models_s N$ 
  using assms

```

```

proof (induct rule: simplify.induct)
  case (tautology-deletion A P)
  then have  $I \models A + \{\#Pos\ P\} + \{\#Neg\ P\}$ 
    by (metis total-over-m-def total-over-set-literal-defined true-cls-singleton true-cls-union
      true-lit-def uminus-Neg union-commute)
  then show ?case by (metis Un-Diff-cancel2 true-clss-singleton true-clss-union)
next
  case (condensation A P)
  then show ?case by (metis Diff-insert-absorb Set.set-insert insertE true-cls-union true-clss-def
    true-clss-singleton true-clss-union)
next
  case (subsumption A B)
  have  $A \neq B$  using subsumption.hyps(2) by auto
  then have  $I \models N - \{B\} \implies I \models A$  using  $\langle A \in N \rangle$  by (simp add: true-clss-def)
  moreover have  $I \models A \implies I \models B$  using  $\langle A < \# B \rangle$  by auto
  ultimately show ?case by (metis insert-Diff-single true-clss-insert)
qed

```

```

lemma simplify-preserves-un-sat:
  fixes  $N\ N' :: 'v\ clauses$ 
  assumes simplify N N'
  and total-over-m I N
  shows  $I \models N \longrightarrow I \models N'$ 
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce+

```

```

lemma simplify-preserves-un-sat'':
  fixes  $N\ N' :: 'v\ clauses$ 
  assumes simplify N N'
  and total-over-m I N'
  shows  $I \models N \longrightarrow I \models N'$ 
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce+

```

```

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

```

```

lemma simplify-preserves-finite:
  assumes simplify  $\psi\ \psi'$ 
  shows finite  $\psi \longleftrightarrow$  finite  $\psi'$ 
  using assms by (induct rule: simplify.induct, auto simp add: remove-def)

```

```

lemma rtrancpl-simplify-preserves-finite:
  assumes rtrancpl simplify  $\psi\ \psi'$ 
  shows finite  $\psi \longleftrightarrow$  finite  $\psi'$ 
  using assms by (induct rule: rtrancpl-induct) (auto simp add: simplify-preserves-finite)

```

```

lemma simplify-atms-of-m:
  assumes simplify  $\psi\ \psi'$ 
  shows  $atms-of-m\ \psi' \subseteq atms-of-m\ \psi$ 
  using assms unfolding atms-of-m-def

```

```

proof (induct rule: simplify.induct)
  case (tautology-deletion A P)
  then show ?case by auto
next
  case (condensation A P)
  moreover have  $A + \{\#P\# \} + \{\#P\# \} \in \psi \implies \exists x \in \psi. \text{atm-of } P \in \text{atm-of } ' \text{set-mset } x$ 
    by (metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff)
  ultimately show ?case by (auto simp add: atms-of-def)
next
  case (subsumption A P)
  then show ?case by auto
qed

lemma rtrancpl-simplify-atms-of-m:
  assumes rtrancpl simplify  $\psi \psi'$ 
  shows  $\text{atms-of-m } \psi' \subseteq \text{atms-of-m } \psi$ 
  using assms apply (induct rule: rtrancpl-induct)
  apply (fastforce intro: simplify-atms-of-m)
  using simplify-atms-of-m by blast

lemma factoring-imp-simplify:
  assumes  $\{\#L\# \} + \{\#L\# \} + C \in N$ 
  shows  $\exists N'. \text{simplify } N N'$ 
proof -
  have  $C + \{\#L\# \} + \{\#L\# \} \in N$  using assms by (simp add: add.commute union-lcomm)
  from condensation[OF this] show ?thesis by blast
qed

```

## 12.2 Unconstrained Resolution

**type-synonym** 'v uncon-state = 'v clauses

**inductive** uncon-res :: 'v uncon-state  $\Rightarrow$  'v uncon-state  $\Rightarrow$  bool **where**

resolution:

$$\{\#Pos \ p\#\} + C \in N \implies \{\#Neg \ p\#\} + D \in N \implies (\{\#Pos \ p\#\} + C, \{\#Neg \ p\#\} + D) \notin \text{already-used}$$

$$\implies \text{uncon-res } (N) (N \cup \{C + D\}) \mid$$

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**  $\psi \in S'$

**using** assms **by** (induct rule: uncon-res.induct) auto

**lemma** rtrancpl-uncon-inference-increasing:

**assumes** rtrancpl uncon-res  $S S'$  **and**  $\psi \in S$

**shows**  $\psi \in S'$

**using** assms **by** (induct rule: rtrancpl-induct) (auto simp add: uncon-res-increasing)

### 12.2.1 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*  $\chi$   $\chi$   
**unfolding** *subsumes-def* **by** *auto*

**lemma** *subsumes-subsumption*:  
**assumes** *subsumes*  $D$   $\chi$   
**and**  $C \subset\# D$  **and**  $\neg$ *tautology*  $\chi$   
**shows** *subsumes*  $C$   $\chi$  **unfolding** *subsumes-def*  
**using** *assms* *subsumption-total-over-m* *subsumption-chained* **unfolding** *subsumes-def*  
**by** (*blast intro!*: *subset-mset.less-imp-le*)

**lemma** *subsumes-tautology*:  
**assumes** *subsumes*  $(C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \})$   $\chi$   
**shows** *tautology*  $\chi$   
**using** *assms* **unfolding** *subsumes-def* **by** (*simp add: tautology-def*)

### 12.3 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**  $\Rightarrow_{Res}$  100) **where**  
*resolution*:  
 $\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$   
*already-used*  
 $\Longrightarrow$  *inference-clause*  $(N, \text{already-used}) (C + D, \text{already-used} \cup \{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\})$  |  
*factoring*:  $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$  *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*)  
 $\Longrightarrow$  *inference*  $S$  (*fst*  $S \cup \{\text{clause}\}$ , *already-used*)

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

**lemma** *inference-clause-preserves-already-used-inv*:  
**assumes** *inference-clause*  $S$   $S'$   
**and** *already-used-inv*  $S$   
**shows** *already-used-inv* (*fst*  $S \cup \{\text{fst } S'\}$ , *snd*  $S'$ )  
**using** *assms* **apply** (*induct rule: inference-clause.induct*)  
**by** *fastforce*+

**lemma** *inference-preserves-already-used-inv*:  
**assumes** *inference*  $S$   $S'$   
**and** *already-used-inv*  $S$   
**shows** *already-used-inv*  $S'$   
**using** *assms*  
**proof** (*induct rule: inference.induct*)  
**case** (*inference-step*  $S$  *clause* *already-used*)  
**then show** ?*case*  
**using** *inference-clause-preserves-already-used-inv*[*of*  $S$  (*clause*, *already-used*)] **by** *simp*

qed

**lemma** *rtrancp-inference-preserves-already-used-inv*:

**assumes** *rtrancp inference S S'*

**and** *already-used-inv S*

**shows** *already-used-inv S'*

**using** *assms apply (induct rule: rtrancp-induct, simp)*

**using** *inference-preserves-already-used-inv unfolding tautology-def by fast*

**lemma** *subsumes-condensation*:

**assumes** *subsumes (C + {#L#} + {#L#}) D*

**shows** *subsumes (C + {#L#}) D*

**using** *assms unfolding subsumes-def by simp*

**lemma** *simplify-preserves-already-used-inv*:

**assumes** *simplify N N'*

**and** *already-used-inv (N, already-used)*

**shows** *already-used-inv (N', already-used)*

**using** *assms*

**proof** (*induct rule: simplify.induct*)

**case** (*condensation C L*)

**then show** *?case*

**using** *subsumes-condensation by simp fast*

**next**

{

**fix** *a:: 'a and A :: 'a set and P*

**have**  $(\exists x \in \text{Set.remove } a \ A. P \ x) \longleftrightarrow (\exists x \in A. x \neq a \wedge P \ x)$  **by** *auto*

} **note** *ex-member-remove = this*

{

**fix** *a a0 :: 'v clause and A :: 'v clauses and y*

**assume** *a ∈ A and a0 ⊂# a*

**then have**  $(\exists x \in A. \text{subsumes } x \ y) \longleftrightarrow (\text{subsumes } a \ y \vee (\exists x \in A. x \neq a \wedge \text{subsumes } x \ y))$

**by** *auto*

} **note** *tt2 = this*

**case** (*subsumption A B*) **note** *A = this(1) and AB = this(2) and B = this(3) and inv = this(4)*

**show** *?case*

**proof** (*standard, standard*)

**fix** *x a b*

**assume** *x: x ∈ snd (N - {B}, already-used) and [simp]: x = (a, b)*

**obtain** *p where p: Pos p ∈# a ∧ Neg p ∈# b and*

*q: (∃χ∈N. subsumes χ (a - {#Pos p#} + (b - {#Neg p#})))*

*∨ tautology (a - {#Pos p#} + (b - {#Neg p#}))*

**using** *inv x by fastforce*

**consider** (*taut*) *tautology (a - {#Pos p#} + (b - {#Neg p#})) |*

*(χ) χ where χ ∈ N subsumes χ (a - {#Pos p#} + (b - {#Neg p#}))*

*¬tautology (a - {#Pos p#} + (b - {#Neg p#}))*

**using** *q by auto*

**then show**

$\exists p. \text{Pos } p \in \# a \wedge \text{Neg } p \in \# b$

$\wedge ((\exists \chi \in \text{fst } (N - \{B\}, \text{already-used}). \text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$

$\vee \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$

**proof** *cases*

**case** *taut*

**then show** *?thesis using p by auto*

**next**



```

      case  $\chi$  note  $H = this$ 
      show ?thesis using  $p \ A \ AB \ B \ subsumes-subsumption[OF - AB H(3)] \ H(1,2)$  by auto
    qed
  qed
next
case (tautology-deletion  $C \ P$ )
then show ?case apply clarify
proof -
  fix  $a \ b$ 
  assume  $C + \{\#Pos \ P\# \} + \{\#Neg \ P\# \} \in N$ 
  assume already-used-inv ( $N$ , already-used)
  and  $(a, b) \in snd \ (N - \{C + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}\}, \textit{already-used})$ 
  then obtain  $p$  where
     $Pos \ p \in \# \ a \wedge Neg \ p \in \# \ b \wedge$ 
     $((\exists \chi \in fst \ (N \cup \{C + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}\}, \textit{already-used}).$ 
       $subsumes \ \chi \ (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \})))$ 
       $\vee \textit{tautology} \ (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \})))$ 
    by fastforce
  moreover have tautology ( $C + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}$ ) by auto
  ultimately show
     $\exists p. Pos \ p \in \# \ a \wedge Neg \ p \in \# \ b$ 
     $\wedge ((\exists \chi \in fst \ (N - \{C + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}\}, \textit{already-used}).$ 
       $subsumes \ \chi \ (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \})))$ 
       $\vee \textit{tautology} \ (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \})))$ 
    by (metis (no-types) Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology
      sup-bot.right-neutral)
  qed
qed

```

**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*:  $atms-of \ (\{\#L\# \} + \{\#L\# \} + C) = atms-of \ (\{\#L\# \} + C)$   
**unfolding** *true-cls-def consistent-interp-def* **by** (*fastforce split: split-if-asm*) +

**lemma** *inference-increasing*:

**assumes** *inference*  $S \ S'$  **and**  $\psi \in fst \ S$   
**shows**  $\psi \in fst \ S'$   
**using** *assms* **by** (*induct rule: inference.induct, auto*)

**lemma** *rtranclp-inference-increasing*:

**assumes** *rtranclp inference*  $S \ S'$  **and**  $\psi \in fst \ S$   
**shows**  $\psi \in fst \ S'$   
**using** *assms* **by** (*induct rule: rtranclp-induct, auto simp add: inference-increasing*)

**lemma** *inference-clause-already-used-increasing*:

**assumes** *inference-clause*  $S \ S'$   
**shows**  $snd \ S \subseteq snd \ S'$   
**using** *assms* **by** (*induct rule: inference-clause.induct, auto*)

**lemma** *inference-already-used-increasing*:

**assumes** *inference*  $S \ S'$

shows  $\text{snd } S \subseteq \text{snd } S'$   
 using **assms apply** (induct rule: inference.induct)  
 using inference-clause-already-used-increasing **by** fastforce

**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  $I \models_s \text{fst } T \longleftrightarrow I \models_s \text{fst } T \cup \{\text{fst } T'\}$   
 using **assms apply** (induct rule: inference-clause.induct)  
 unfolding consistent-interp-def true-clss-def **by** auto force+

**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  $I \models_s \text{fst } T \longleftrightarrow I \models_s \text{fst } T'$   
 using **assms apply** (induct rule: inference.induct)  
 using inference-clause-preserves-un-sat **by** fastforce

**lemma** inference-clause-preserves-atms-of-m:  
 assumes inference-clause  $S \ S'$   
 shows  $\text{atms-of-m } (\text{fst } (\text{fst } S \cup \{\text{fst } S'\}), \text{snd } S') \subseteq \text{atms-of-m } (\text{fst } S)$   
 using **assms apply** (induct rule: inference-clause.induct)  
 apply auto  
 apply (metis Set.set-insert UnCI atms-of-m-insert atms-of-plus)  
 apply (metis Set.set-insert UnCI atms-of-m-insert atms-of-plus)  
 apply (simp add: in-m-in-literals union-assoc)  
 unfolding atms-of-m-def **using** **assms** **by** fastforce

**lemma** inference-preserves-atms-of-m:  
 fixes  $N \ N' :: 'v \text{ clauses}$   
 assumes inference  $T \ T'$   
 shows  $\text{atms-of-m } (\text{fst } T') \subseteq \text{atms-of-m } (\text{fst } T)$   
 using **assms apply** (induct rule: inference.induct)  
 using inference-clause-preserves-atms-of-m **by** fastforce

**lemma** inference-preserves-total:  
 fixes  $N \ N' :: 'v \text{ clauses}$   
 assumes inference  $(N, \text{already-used}) \ (N', \text{already-used}')$   
 shows  $\text{total-over-m } I \ N \Longrightarrow \text{total-over-m } I \ N'$   
 using **assms** inference-preserves-atms-of-m **unfolding** total-over-m-def total-over-set-def  
**by** fastforce

**lemma** rtranclp-inference-preserves-total:  
 assumes rtranclp inference  $T \ T'$   
 shows  $\text{total-over-m } I \ (\text{fst } T) \Longrightarrow \text{total-over-m } I \ (\text{fst } T')$   
 using **assms** **by** (induct rule: rtranclp-induct, auto simp add: inference-preserves-total)

**lemma** rtranclp-inference-preserves-un-sat:  
 assumes rtranclp inference  $N \ N'$

**and** *total-over-m* *I* (*fst* *N*)  
**and** *consistent*: *consistent-interp* *I*  
**shows**  $I \models_s \text{fst } N \longleftrightarrow I \models_s \text{fst } N'$   
**using** *assms* **apply** (*induct* rule: *rtranclp-induct*)  
**apply** (*simp* add: *inference-preserves-un-sat*)  
**using** *inference-preserves-un-sat* *rtranclp-inference-preserves-total* **by** *blast*

**lemma** *inference-preserves-finite*:  
**assumes** *inference*  $\psi \psi'$  **and** *finite* (*fst*  $\psi$ )  
**shows** *finite* (*fst*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *inference.induct*, *auto* *simp* add: *simplify-preserves-finite*)

**lemma** *inference-clause-preserves-finite-snd*:  
**assumes** *inference-clause*  $\psi \psi'$  **and** *finite* (*snd*  $\psi$ )  
**shows** *finite* (*snd*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *inference-clause.induct*, *auto*)

**lemma** *inference-preserves-finite-snd*:  
**assumes** *inference*  $\psi \psi'$  **and** *finite* (*snd*  $\psi$ )  
**shows** *finite* (*snd*  $\psi'$ )  
**using** *assms* *inference-clause-preserves-finite-snd* **by** (*induct* rule: *inference.induct*, *fastforce*)

**lemma** *rtranclp-inference-preserves-finite*:  
**assumes** *rtranclp* *inference*  $\psi \psi'$  **and** *finite* (*fst*  $\psi$ )  
**shows** *finite* (*fst*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *rtranclp-induct*)  
*(auto simp add: simplify-preserves-finite inference-preserves-finite)*

**lemma** *consistent-interp-insert*:  
**assumes** *consistent-interp* *I*  
**and** *atm-of* *P*  $\notin$  *atm-of* ' *I*  
**shows** *consistent-interp* (*insert* *P* *I*)  
**proof** –  
**have** *P*: *insert* *P* *I* = *I*  $\cup$  {*P*} **by** *auto*  
**show** ?thesis **unfolding** *P*  
**apply** (rule *consistent-interp-disjoint*)  
**using** *assms* **by** (*auto simp* add: *atms-of-s-def*)  
**qed**

**lemma** *simplify-clause-preserves-sat*:  
**assumes** *simp*: *simplify*  $\psi \psi'$   
**and** *satisfiable*  $\psi'$   
**shows** *satisfiable*  $\psi$   
**using** *assms*  
**proof** *induction*  
**case** (*tautology-deletion* *A* *P*) **note** *AP* = *this*(1) **and** *sat* = *this*(2)  
**let** ?*A*' = *A* + {#*Pos* *P*#} + {#*Neg* *P*#}  
**let** ? $\psi'$  =  $\psi - \{?A\}$   
**obtain** *I* **where**  
*I*:  $I \models_s ?\psi'$  **and**  
*cons*: *consistent-interp* *I* **and**  
*tot*: *total-over-m* *I* ? $\psi'$

```

    using sat unfolding satisfiable-def by auto
  { assume Pos P ∈ I ∨ Neg P ∈ I
    then have I ⊨ ?A' by auto
    then have I ⊨s ψ using I by (metis insert-Diff tautology-deletion.hyps true-clss-insert)
    then have ?case using cons tot by auto
  }
  moreover {
    assume Pos: Pos P ∉ I and Neg: Neg P ∉ I
    then have consistent-interp (I ∪ {Pos P}) using cons by simp
    moreover have I'A: I ∪ {Pos P} ⊨ ?A' by auto
    have {Pos P} ∪ I ⊨s ψ - {A + {#Pos P#} + {#Neg P#}}
      using ⟨I ⊨s ψ - {A + {#Pos P#} + {#Neg P#}}⟩ true-clss-union-increase' by blast
    then have I ∪ {Pos P} ⊨s ψ
      by (metis (no-types) Un-empty-right Un-insert-left Un-insert-right I'A insert-Diff
        sup-bot.left-neutral tautology-deletion.hyps true-clss-insert)
    ultimately have ?case using satisfiable-carac' by blast
  }
  ultimately show ?case by blast
next
case (condensation A L) note AL = this(1) and sat = this(2)
have f3: simplify ψ (ψ - {A + {#L#} + {#L#}} ∪ {A + {#L#}})
  using AL simplify.condensation by blast
obtain LL :: 'a literal multiset set ⇒ 'a literal set where
  f4: LL (ψ - {A + {#L#} + {#L#}} ∪ {A + {#L#}}) ⊨s ψ - {A + {#L#} + {#L#}} ∪ {A
+ {#L#}}
  ∧ consistent-interp (LL (ψ - {A + {#L#} + {#L#}} ∪ {A + {#L#}}))
  ∧ total-over-m (LL (ψ - {A + {#L#} + {#L#}}
    ∪ {A + {#L#}})) (ψ - {A + {#L#} + {#L#}} ∪ {A + {#L#}})
  using sat by (meson satisfiable-def)
have f5: insert (A + {#L#} + {#L#}) (ψ - {A + {#L#} + {#L#}}) = ψ
  using AL by fastforce
have atms-of (A + {#L#} + {#L#}) = atms-of ({#L#} + A)
  by simp
then show ?case
  using f5 f4 f3 by (metis (no-types) add.commute satisfiable-def simplify-preserves-un-sat'
    total-over-m-insert total-over-m-union)
next
case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and sat = this(4)
let ?ψ' = ψ - {B}
obtain I where I: I ⊨s ?ψ' and cons: consistent-interp I and tot: total-over-m I ?ψ'
  using sat unfolding satisfiable-def by auto
have I ⊨ A using A I by (metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def)
then have I ⊨ B using AB subset-mset.less-imp-le true-clss-mono-leD by blast
then have I ⊨s ψ using I by (metis insert-Diff-single true-clss-insert)
then show ?case using cons satisfiable-carac' by blast
qed

lemma simplify-preserves-unsat:
  assumes inference ψ ψ'
  shows satisfiable (fst ψ') ⟶ satisfiable (fst ψ)
  using assms apply (induct rule: inference.induct)
  using satisfiable-decreasing by (metis fst-conv)+

lemma inference-preserves-unsat:
  assumes inference** S S'

```

**shows** *satisfiable* (*fst S'*)  $\longrightarrow$  *satisfiable* (*fst S*)  
**using** *assms* **apply** (*induct rule: rtrancpl-induct*)  
**apply** *simp-all*  
**using** *simplify-preserves-unsat* **by** *blast*

**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]*:  
 $(\bigwedge xs:: 'v \text{ sem-tree. } (\bigwedge ys:: 'v \text{ sem-tree. } \text{sem-tree-size } ys < \text{sem-tree-size } xs \implies P \text{ } ys) \implies P \text{ } xs)$   
 $\implies P \text{ } xs$   
**by** (*fact Nat.measure-induct-rule*)

**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  $\psi$*   $\longleftrightarrow$   
 $(\text{partial-interps } ag (I \cup \{ \text{Pos } v \}) \psi \wedge \text{partial-interps } ad (I \cup \{ \text{Neg } v \}) \psi)$

**lemma** *simplify-preserve-partial-leaf*:  
*simplify N N'  $\implies$  partial-interps Leaf I N  $\implies$  partial-interps Leaf I N'*  
**apply** (*induct rule: simplify.induct*)  
**using** *union-lcomm* **apply** *auto[1]*  
**apply** (*simp, metis atms-of-plus total-over-set-union true-cls-union*)  
**apply** *simp*  
**by** (*metis atms-of-m-singleton mset-le-exists-conv subset-mset-def true-cls-mono-leD total-over-m-def total-over-m-sum*)

**lemma** *simplify-preserve-partial-tree*:  
**assumes** *simplify N N'*  
**and** *partial-interps t I N*  
**shows** *partial-interps t I N'*  
**using** *assms* **apply** (*induct t arbitrary: I, simp*)  
**using** *simplify-preserve-partial-leaf* **by** *metis*

**lemma** *inference-preserve-partial-tree*:  
**assumes** *inference S S'*  
**and** *partial-interps t I (fst S)*  
**shows** *partial-interps t I (fst S')*  
**using** *assms* **apply** (*induct t arbitrary: I, simp-all*)  
**by** (*meson inference-increasing*)

**lemma** *rtrancpl-inference-preserve-partial-tree*:  
**assumes** *rtrancpl inference N N'*  
**and** *partial-interps t I (fst N)*  
**shows** *partial-interps t I (fst N')*  
**using** *assms* **apply** (*induct rule: rtrancpl-induct, auto*)  
**using** *inference-preserve-partial-tree* **by** *force*

```

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)  $\psi$ ))
by auto
termination
  apply (relation measure ( $\lambda(A, -).$  card A), simp-all)
  apply (metis Min-in card-Diff1-less remove-def)+
done
declare build-sem-tree.induct[case-names tree]

lemma unsatisfiable-empty[simp]:
   $\neg$ unsatisfiable {}
  unfolding satisfiable-def apply auto
  using consistent-interp-def unfolding total-over-m-def total-over-set-def atms-of-m-def by blast

lemma partial-interps-build-sem-tree-atms-general:
  fixes  $\psi :: 'v :: linorder$  clauses and  $p :: 'v$  literal list
  assumes unsat: unsatisfiable  $\psi$  and finite  $\psi$  and consistent-interp I
  and finite atms
  and atms-of-m  $\psi$  = atms  $\cup$  atms-of-s I and atms  $\cap$  atms-of-s I = {}
  shows partial-interps (build-sem-tree atms  $\psi$ ) I  $\psi$ 
  using assms
proof (induct arbitrary: I rule: build-sem-tree.induct)
case (1 atms  $\psi$  Ia) note IH1 = this(1) and IH2 = this(2) and unsat = this(3) and finite = this(4)
  and cons = this(5) and f = this(6) and un = this(7) and disj = this(8)
  {
    assume atms: atms = {}
    then have atmsIa: atms-of-m  $\psi$  = atms-of-s Ia using un by auto
    then have total-over-m Ia  $\psi$  unfolding total-over-m-def atmsIa by auto
    then have  $\chi$ :  $\exists \chi \in \psi. \neg Ia \models \chi$ 
      using unsat cons unfolding true-clss-def satisfiable-def by auto
    then have build-sem-tree atms  $\psi$  = Leaf using atms by auto
    moreover
      have tot:  $\bigwedge \chi. \chi \in \psi \implies$  total-over-m Ia { $\chi$ }
      unfolding total-over-m-def total-over-set-def atms-of-m-def atms-of-s-def
      using atmsIa atms-of-m-def by fastforce
    have partial-interps Leaf Ia  $\psi$ 
      using  $\chi$  tot by (auto simp add: total-over-m-def total-over-set-def atms-of-m-def)

    ultimately have ?case by metis
  }
moreover {
  assume atms: atms  $\neq$  {}
  have build-sem-tree atms  $\psi$  = Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    using build-sem-tree.simps[of atms  $\psi$ ] f atms by metis

  have consistent-interp (Ia  $\cup$  {Pos (Min atms)}) unfolding consistent-interp-def
    by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
      f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2))

```

```

    uminus-Neg uminus-Pos)
  moreover have atms-of-m  $\psi = \text{Set.remove } (\text{Min atms}) \text{ atms} \cup \text{atms-of-s } (Ia \cup \{\text{Pos } (\text{Min atms})\})$ 
    using Min-in atms f un by fastforce
  moreover have disj':  $\text{Set.remove } (\text{Min atms}) \text{ atms} \cap \text{atms-of-s } (Ia \cup \{\text{Pos } (\text{Min atms})\}) = \{\}$ 
    by simp (metis disj disjoint-iff-not-equal member-remove)
  moreover have finite (Set.remove (Min atms) atms) using f by (simp add: remove-def)
  ultimately have subtree1: partial-interps (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    ( $Ia \cup \{\text{Pos } (\text{Min atms})\}$ )  $\psi$ 
    using IH1[of  $Ia \cup \{\text{Pos } (\text{Min atms})\}$ ] atms f unsat finite by metis

  have consistent-interp ( $Ia \cup \{\text{Neg } (\text{Min atms})\}$ ) unfolding consistent-interp-def
    by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
      f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
      uminus-Neg)
  moreover have atms-of-m  $\psi = \text{Set.remove } (\text{Min atms}) \text{ atms} \cup \text{atms-of-s } (Ia \cup \{\text{Neg } (\text{Min atms})\})$ 
    using (atms-of-m  $\psi = \text{Set.remove } (\text{Min atms}) \text{ atms} \cup \text{atms-of-s } (Ia \cup \{\text{Pos } (\text{Min atms})\})$ ) by blast

  moreover have disj':  $\text{Set.remove } (\text{Min atms}) \text{ atms} \cap \text{atms-of-s } (Ia \cup \{\text{Neg } (\text{Min atms})\}) = \{\}$ 
    using disj by auto
  moreover have finite (Set.remove (Min atms) atms) using f by (simp add: remove-def)
  ultimately have subtree2: partial-interps (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    ( $Ia \cup \{\text{Neg } (\text{Min atms})\}$ )  $\psi$ 
    using IH2[of  $Ia \cup \{\text{Neg } (\text{Min atms})\}$ ] atms f unsat finite by metis

  then have ?case
    using IH1 subtree1 subtree2 f local.finite unsat atms by simp
}
ultimately show ?case by metis
qed

```

```

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-m  $\psi$ )  $\psi$ )  $\{\}$   $\psi$ 
proof -
  have consistent-interp  $\{\}$  unfolding consistent-interp-def by auto
  moreover have atms-of-m  $\psi = \text{atms-of-m } \psi \cup \text{atms-of-s } \{\}$  unfolding atms-of-s-def by auto
  moreover have  $\text{atms-of-m } \psi \cap \text{atms-of-s } \{\} = \{\}$  unfolding atms-of-s-def by auto
  moreover have finite (atms-of-m  $\psi$ ) unfolding atms-of-m-def using finite by simp
  ultimately show partial-interps (build-sem-tree (atms-of-m  $\psi$ )  $\psi$ )  $\{\}$   $\psi$ 
    using partial-interps-build-sem-tree-atms-general[of  $\psi \{\}$  atms-of-m  $\psi$ ] assms by metis
qed

```

```

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  $\exists \psi' \chi'. \text{inference}^{**} \psi \psi' \wedge \chi' \in \text{fst } \psi' \wedge (\forall L. L \in \# \chi \longleftrightarrow L \in \# \chi')$ 
     $\wedge \text{count } \chi' L = 1$ 
     $\wedge (\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi')$ 
     $\wedge (I \models \chi \longleftrightarrow I \models \chi')$ 
     $\wedge (\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi'\})$ 
  using assms
proof (induct n arbitrary:  $\chi \psi$ )

```

```

case 0
then show ?case by simp
next
case (Suc n χ)
note IH = this(1) and count = this(2) and L = this(3) and χ = this(4)
{
  assume n = 0
  then have inference** ψ ψ
  and χ ∈ fst ψ
  and ∀ L. (L ∈# χ) ⟷ (L ∈# χ)
  and count χ L = (1::nat)
  and ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ
    by (auto simp add: count L χ)
  then have ?case by metis
}
moreover {
  assume n > 0
  then have ∃ C. χ = C + {#L, L#}
    by (metis L One-nat-def add-diff-cancel-right' count-diff count-single diff-Suc-Suc diff-zero
      local.count multi-member-split union-assoc)
  then obtain C where C: χ = C + {#L, L#} by metis
  let ?χ' = C + {#L#}
  let ?ψ' = (fst ψ ∪ {?χ'}, snd ψ)
  have φ: ∀ φ ∈ fst ψ. (φ ∈ fst ψ ∨ φ ≠ ?χ') ⟷ φ ∈ fst ?ψ' unfolding C by auto
  have inf: inference ψ ?ψ'
    using C factoring χ prod.collapse union-commute inference-step by metis
  moreover have count': count ?χ' L = n using C count by auto
  moreover have Lχ': L :# ?χ' by auto
  moreover have χ'ψ': ?χ' ∈ fst ?ψ' by auto
  ultimately obtain ψ'' and χ''
  where
    inference** ?ψ' ψ'' and
    α: χ'' ∈ fst ψ'' and
    ∀ La. (La ∈# ?χ') ⟷ (La ∈# χ'') and
    β: count χ'' L = (1::nat) and
    φ': ∀ φ. φ ∈ fst ?ψ' ⟶ φ ∈ fst ψ'' and
    Iχ: I ⊨ ?χ' ⟷ I ⊨ χ'' and
    tot: ∀ I'. total-over-m I' {?χ'} ⟶ total-over-m I' {χ''}
    using IH[of ?χ' ?ψ'] count' Lχ' χ'ψ' by blast

  then have inference** ψ ψ''
  and ∀ La. (La ∈# χ) ⟷ (La ∈# χ'')
  using inf unfolding C by auto
  moreover have ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ'' using φ φ' by metis
  moreover have I ⊨ χ ⟷ I ⊨ χ'' using Iχ unfolding true-cls-def C by auto
  moreover have ∀ I'. total-over-m I' {χ} ⟶ total-over-m I' {χ''}
    using tot unfolding C total-over-m-def by auto
  ultimately have ?case using φ φ' α β by metis
}
ultimately show ?case by auto
qed

```

**lemma** *can-decrease-tree-size:*

fixes ψ :: 'v state and tree :: 'v sem-tree  
 assumes finite (fst ψ) and already-used-inv ψ



```

and partial-interps tree I (fst  $\psi$ )
shows  $\exists (tree':: 'v \text{ sem-tree}) \psi'. \text{inference}^{**} \psi \psi' \wedge \text{partial-interps tree}' I (\text{fst } \psi')$ 
 $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$ 
using assms
proof (induct arbitrary: I rule: sem-tree-size)
case (bigger xs I) note IH = this(1) and finite = this(2) and a-u-i = this(3) and part = this(4)

{
  assume sem-tree-size xs = 0
  then have ?case using part by blast
}

moreover {
  assume sn0: sem-tree-size xs > 0
  obtain ag ad v where xs: xs = Node v ag ad using sn0 by (case-tac xs, auto)
  {
    assume sem-tree-size ag = 0 and sem-tree-size ad = 0
    then have ag: ag = Leaf and ad: ad = Leaf by (case-tac ag, auto) (case-tac ad, auto)

    then obtain  $\chi \chi'$  where
       $\chi: \neg I \cup \{Pos\ v\} \models \chi$  and
       $\text{tot}\chi: \text{total-over-m } (I \cup \{Pos\ v\}) \{\chi\}$  and
       $\chi\psi: \chi \in \text{fst } \psi$  and
       $\chi': \neg I \cup \{Neg\ v\} \models \chi'$  and
       $\text{tot}\chi': \text{total-over-m } (I \cup \{Neg\ v\}) \{\chi'\}$  and
       $\chi'\psi: \chi' \in \text{fst } \psi$ 
    using part unfolding xs by auto
    have Posv:  $\neg Pos\ v \in \# \chi$  using  $\chi$  unfolding true-cls-def true-lit-def by auto
    have Negv:  $\neg Neg\ v \in \# \chi'$  using  $\chi'$  unfolding true-cls-def true-lit-def by auto
    {
      assume Neg $\chi$ :  $\neg Neg\ v \in \# \chi$ 
      have  $\neg I \models \chi$  using  $\chi$  Posv unfolding true-cls-def true-lit-def by auto
      moreover have  $\text{total-over-m } I \{\chi\}$ 
        using Posv Neg $\chi$  atm-imp-pos-or-neg-lit tot $\chi$  unfolding total-over-m-def total-over-set-def
        by fastforce
      ultimately have partial-interps Leaf I (fst  $\psi$ )
      and sem-tree-size Leaf < sem-tree-size xs
      and inference**  $\psi \psi$ 
      unfolding xs by (auto simp add:  $\chi\psi$ )
    }
  }
  moreover {
    assume Pos $\chi$ :  $\neg Pos\ v \in \# \chi'$ 
    then have I $\chi$ :  $\neg I \models \chi'$  using  $\chi'$  Posv unfolding true-cls-def true-lit-def by auto
    moreover have  $\text{total-over-m } I \{\chi'\}$ 
      using Negv Pos $\chi$  atm-imp-pos-or-neg-lit tot $\chi'$ 
      unfolding total-over-m-def total-over-set-def by fastforce
    ultimately have partial-interps Leaf I (fst  $\psi$ ) and
      sem-tree-size Leaf < sem-tree-size xs and
      inference**  $\psi \psi$ 
    using  $\chi'\psi$  I $\chi$  unfolding xs by auto
  }
}
moreover {
  assume neg:  $Neg\ v \in \# \chi$  and pos:  $Pos\ v \in \# \chi'$ 
  then obtain  $\psi' \chi_2$  where inf: rtrancp inference  $\psi \psi'$  and  $\chi_2\text{incl}: \chi_2 \in \text{fst } \psi'$ 
  and  $\chi\chi_2\text{-incl}: \forall L. L : \# \chi \longleftrightarrow L : \# \chi_2$ 

```

```

and count $\chi 2$ : count  $\chi 2$  (Neg  $v$ ) = 1
and  $\varphi$ :  $\forall \varphi::'v$  literal multiset.  $\varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi'$ 
and  $I\chi$ :  $I \models \chi \longleftrightarrow I \models \chi 2$ 
and tot-imp $\chi$ :  $\forall I'$ . total-over-m  $I' \{ \chi \} \longrightarrow \text{total-over-m } I' \{ \chi 2 \}$ 
using can-decrease-count[of  $\chi$  Neg  $v$  count  $\chi$  (Neg  $v$ )  $\psi$   $I$ ]  $\chi \psi \chi' \psi$  by auto

have  $\chi' \in \text{fst } \psi'$  by (simp add:  $\chi' \psi \varphi$ )
with pos
obtain  $\psi'' \chi 2'$  where
  inf': inference**  $\psi' \psi''$ 
and  $\chi 2'$ -incl:  $\chi 2' \in \text{fst } \psi''$ 
and  $\chi' \chi 2$ -incl:  $\forall L::'v$  literal.  $(L \in \# \chi') = (L \in \# \chi 2')$ 
and count $\chi 2'$ : count  $\chi 2'$  (Pos  $v$ ) = (1::nat)
and  $\varphi'$ :  $\forall \varphi::'v$  literal multiset.  $\varphi \in \text{fst } \psi' \longrightarrow \varphi \in \text{fst } \psi''$ 
and  $I\chi'$ :  $I \models \chi' \longleftrightarrow I \models \chi 2'$ 
and tot-imp $\chi'$ :  $\forall I'$ . total-over-m  $I' \{ \chi' \} \longrightarrow \text{total-over-m } I' \{ \chi 2' \}$ 
using can-decrease-count[of  $\chi' \text{ Pos } v$  count  $\chi' \text{ (Pos } v) \psi' I$ ] by auto

obtain  $C$  where  $\chi 2$ :  $\chi 2 = C + \{ \# \text{Neg } v \# \}$  and neg $C$ : Neg  $v \notin \# C$  and pos $C$ : Pos  $v \notin \# C$ 
by (metis (no-types, lifting) One-nat-def Posv Suc-inject Suc-pred  $\chi \chi 2$ -incl count $\chi 2$ 
  count-diff count-single gr0I insert-DiffM insert-DiffM2 multi-member-skip
  old.nat.distinct(2))

obtain  $C'$  where
   $\chi 2'$ :  $\chi 2' = C' + \{ \# \text{Pos } v \# \}$  and
  pos $C'$ : Pos  $v \notin \# C'$  and
  neg $C'$ : Neg  $v \notin \# C'$ 
proof -
  assume a1:  $\bigwedge C'. \llbracket \chi 2' = C' + \{ \# \text{Pos } v \# \}; \text{Pos } v \notin \# C'; \text{Neg } v \notin \# C' \rrbracket \Longrightarrow \text{thesis}$ 
  have f2:  $\bigwedge n. (n::\text{nat}) - n = 0$ 
    by simp
  have Neg  $v \notin \# \chi 2' - \{ \# \text{Pos } v \# \}$ 
    using Negv  $\chi' \chi 2$ -incl by auto
  then show ?thesis
    using f2 a1 by (metis add commute count $\chi 2'$  count-diff count-single insert-DiffM
      less-nat-zero-code zero-less-one)
qed

have already-used-inv  $\psi'$ 
  using rtrancplp-inference-preserves-already-used-inv[of  $\psi \psi'$ ] a-u-i inf by blast
then have a-u-i- $\psi''$ : already-used-inv  $\psi''$ 
  using rtrancplp-inference-preserves-already-used-inv a-u-i inf' unfolding tautology-def
  by simp

have tot $C$ : total-over-m  $I \{ C \}$ 
  using tot-imp $\chi$  tot $\chi$  tot-over-m-remove[of  $I \text{ Pos } v C$ ] neg $C$  pos $C$  unfolding  $\chi 2$ 
  by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
have tot $C'$ : total-over-m  $I \{ C' \}$ 
  using tot-imp $\chi'$  tot $\chi'$  total-over-m-sum tot-over-m-remove[of  $I \text{ Neg } v C'$ ] neg $C'$  pos $C'$ 
  unfolding  $\chi 2'$  by (metis total-over-m-sum uminus-Neg)
have  $\neg I \models C + C'$ 
  using  $\chi I\chi \chi' I\chi'$  unfolding  $\chi 2 \chi 2'$  true-cls-def Bex-mset-def
  by (metis add-gr-0 count-union true-cls-singleton true-cls-union-increase)
then have part-I- $\psi'''$ : partial-interps Leaf  $I$  (fst  $\psi'' \cup \{ C + C' \}$ )
  using tot $C$  tot $C'$  by simp

```

```

(metis  $\hookrightarrow I \models C + C'$ ) atms-of-m-singleton total-over-m-def total-over-m-sum)
{
  assume ( $\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C \} \notin \text{snd } \psi''$ )
  then have inf'': inference  $\psi''$  (fst  $\psi'' \cup \{C + C'\}$ , snd  $\psi'' \cup \{(\chi 2', \chi 2)\}$ )
    using add.commute  $\varphi' \chi 2 \text{incl } \{ \chi 2' \in \text{fst } \psi'' \}$  unfolding  $\chi 2 \chi 2'$ 
    by (metis prod.collapse inference-step resolution)
  have inference**  $\psi$  (fst  $\psi'' \cup \{C + C'\}$ , snd  $\psi'' \cup \{(\chi 2', \chi 2)\}$ )
    using inf inf' inf'' rtranclp-trans by auto
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case using part-I- $\psi'''$  by (metis fst-conv)
}
moreover {
  assume a: ( $\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C \} \in \text{snd } \psi''$ )
  then have ( $\exists \chi \in \text{fst } \psi''$ . ( $\forall I$ . total-over-m  $I \{C + C'\} \longrightarrow \text{total-over-m } I \{\chi\}$ )
     $\wedge (\forall I$ . total-over-m  $I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ )
     $\vee$  tautology ( $C' + C$ )
  proof -
    obtain p where p:  $Pos\ p \in \# (\{\#Pos\ v\# \} + C')$  and
      n:  $Neg\ p \in \# (\{\#Neg\ v\# \} + C)$  and
      decomp: ( $\exists \chi \in \text{fst } \psi''$ .
        ( $\forall I$ . total-over-m  $I \{(\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \}$ 
           $+ ((\{\#Neg\ v\# \} + C) - \{\#Neg\ p\# \})\}$ 
           $\longrightarrow \text{total-over-m } I \{\chi\}$ )
           $\wedge (\forall I$ . total-over-m  $I \{\chi\} \longrightarrow I \models \chi$ 
           $\longrightarrow I \models (\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \} + ((\{\#Neg\ v\# \} + C) - \{\#Neg\ p\# \})$ 
          )
           $\vee$  tautology ( $(\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \} + ((\{\#Neg\ v\# \} + C) - \{\#Neg\ p\# \}))$ )
        )
      using a by (blast intro: allE[OF a-u-i- $\psi''$ ][unfolding subsumes-def Ball-def],
        of ( $\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C$ ))
    { assume  $p \neq v$ 
      then have  $Pos\ p \in \# C' \wedge Neg\ p \in \# C$  using p n by force
      then have ?thesis by (metis add-gr-0 count-union tautology-Pos-Neg)
    }
  moreover {
    assume  $p = v$ 
    then have ?thesis using decomp by (metis add.commute add-diff-cancel-left')
  }
  ultimately show ?thesis by auto
}
qed
moreover {
  assume  $\exists \chi \in \text{fst } \psi''$ . ( $\forall I$ . total-over-m  $I \{C + C'\} \longrightarrow \text{total-over-m } I \{\chi\}$ )
     $\wedge (\forall I$ . total-over-m  $I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C$ )
  then obtain  $\vartheta$  where  $\vartheta$ :  $\vartheta \in \text{fst } \psi''$  and
    tot- $\vartheta$ -CC':  $\forall I$ . total-over-m  $I \{C + C'\} \longrightarrow \text{total-over-m } I \{\vartheta\}$  and
     $\vartheta$ -inv:  $\forall I$ . total-over-m  $I \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$  by blast
  have partial-interps Leaf I (fst  $\psi''$ )
    using tot- $\vartheta$ -CC'  $\vartheta$ -inv totC totC'  $\hookrightarrow I \models C + C'$  total-over-m-sum by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case by (metis inf inf' rtranclp-trans)
}
moreover {
  assume tautCC': tautology ( $C' + C$ )
  have total-over-m  $I \{C' + C\}$  using totC totC' total-over-m-sum by auto
  then have  $\neg \text{tautology } (C' + C)$ 
    using  $\hookrightarrow I \models C + C'$  unfolding add.commute[of C C'] total-over-m-def

```

```

      unfolding tautology-def by auto
      then have False using tautCC' unfolding tautology-def by auto
    }
    ultimately have ?case by auto
  }
  ultimately have ?case by auto
}
ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
  assume size-ag: sem-tree-size ag > 0
  have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
  moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
    and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ag < sem-tree-size xs → finite (fst ψ) → already-used-inv ψ
    → ( partial-interps ag (I ∪ {Pos v}) (fst ψ) →
      (∃ tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
        ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)))
    using IH by auto
  ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
    inf: inference** ψ ψ'
    and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
    and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
    using finite part rtranclp.rtrancl-refl a-u-i by blast

  have partial-interps ad (I ∪ {Neg v}) (fst ψ')
    using rtranclp-inference-preserve-partial-tree inf partad by metis
  then have partial-interps (Node v tree' ad) I (fst ψ') using part by auto
  then have ?case using inf size size-ag part unfolding xs by fastforce
}
moreover {
  assume size-ad: sem-tree-size ad > 0
  have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
  moreover have partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
    partial-interps ad (I ∪ {Neg v}) (fst ψ)
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ad < sem-tree-size xs → finite (fst ψ) → already-used-inv ψ
    → ( partial-interps ad (I ∪ {Neg v}) (fst ψ)
      → (∃ tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Neg v}) (fst ψ')
        ∧ (sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0)))
    using IH by auto
  ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
    inf: inference** ψ ψ'
    and part: partial-interps tree' (I ∪ {Neg v}) (fst ψ')
    and size: sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0
    using finite part rtranclp.rtrancl-refl a-u-i by blast

  have partial-interps ag (I ∪ {Pos v}) (fst ψ')
    using rtranclp-inference-preserve-partial-tree inf partag by metis
  then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
  then have ?case using inf size size-ad unfolding xs by fastforce
}
ultimately have ?case by auto
}

```

ultimately show ?case by auto  
qed

**lemma** *inference-completeness-inv*:

**fixes**  $\psi :: 'v :: \text{linorder state}$

**assumes**

*unsat*:  $\neg \text{satisfiable (fst } \psi)$  **and**

*finite*:  $\text{finite (fst } \psi)$  **and**

*a-u-v*: *already-used-inv*  $\psi$

**shows**  $\exists \psi'. (\text{inference}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

**proof** –

**obtain** *tree* **where** *partial-interps* *tree*  $\{\}$  (*fst*  $\psi$ )

**using** *partial-interps-build-sem-tree-atms* *assms* **by** *metis*

**then show** ?thesis

**using** *unsat* *finite* *a-u-v*

**proof** (*induct tree arbitrary*:  $\psi$  *rule*: *sem-tree-size*)

**case** (*bigger tree*  $\psi$ ) **note**  $H = \text{this}$

{

**fix**  $\chi$

**assume** *tree*: *tree* = *Leaf*

**obtain**  $\chi$  **where**  $\chi: \neg \{\} \models \chi$  **and** *tot* $\chi$ : *total-over-m*  $\{\} \{\chi\}$  **and**  $\chi\psi: \chi \in \text{fst } \psi$

**using**  $H$  *unfolding* *tree* **by** *auto*

**moreover have**  $\{\#\} = \chi$

**using** *tot* $\chi$  *unfolding* *total-over-m-def* *total-over-set-def* **by** *fastforce*

**moreover have** *inference*<sup>\*\*</sup>  $\psi \psi$  **by** *auto*

**ultimately have** ?case **by** *metis*

}

**moreover** {

**fix**  $v$  *tree1* *tree2*

**assume** *tree*: *tree* = *Node*  $v$  *tree1* *tree2*

**obtain**

*tree'*  $\psi'$  **where** *inf*: *inference*<sup>\*\*</sup>  $\psi \psi'$  **and**

*part'*: *partial-interps* *tree'*  $\{\}$  (*fst*  $\psi'$ ) **and**

*decrease*: *sem-tree-size* *tree'* < *sem-tree-size* *tree*  $\vee$  *sem-tree-size* *tree* = 0

**using** *can-decrease-tree-size*[*of*  $\psi$ ]  $H(2,4,5)$  *unfolding* *tautology-def* **by** *meson*

**have** *sem-tree-size* *tree'* < *sem-tree-size* *tree* **using** *decrease* *unfolding* *tree* **by** *auto*

**moreover have** *finite* (*fst*  $\psi'$ ) **using** *rtranclp-inference-preserves-finite* *inf*  $H(4)$  **by** *metis*

**moreover have** *unsatisfiable* (*fst*  $\psi'$ )

**using** *inference-preserves-unsat* *inf* *bigger.prem*s(2) **by** *blast*

**moreover have** *already-used-inv*  $\psi'$

**using**  $H(5)$  *inf* *rtranclp-inference-preserves-already-used-inv*[*of*  $\psi \psi'$ ] **by** *auto*

**ultimately have** ?case **using** *inf* *rtranclp-trans* *part'*  $H(1)$  **by** *fastforce*

}

**ultimately show** ?case **by** (*case-tac* *tree*, *auto*)

qed

qed

**lemma** *inference-completeness*:

**fixes**  $\psi :: 'v :: \text{linorder state}$

**assumes** *unsat*:  $\neg \text{satisfiable (fst } \psi)$

**and** *finite*: *finite* (*fst*  $\psi$ )

**and** *snd*  $\psi = \{\}$

**shows**  $\exists \psi'. (\text{rtranclp inference } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

**proof** –

**have** *already-used-inv*  $\psi$  *unfolding* *assms* **by** *auto*

then show *?thesis* using *assms inference-completeness-inv* by *blast*  
qed

lemma *inference-soundness*:

fixes  $\psi :: 'v :: \text{linorder state}$

assumes *rtranclp inference*  $\psi$   $\psi'$  and  $\{\#\} \in \text{fst } \psi'$

shows *unsatisfiable* (*fst*  $\psi$ )

using *assms* by (*meson rtranclp-inference-preserves-un-sat satisfiable-def true-cls-empty true-cls-def*)

lemma *inference-soundness-and-completeness*:

fixes  $\psi :: 'v :: \text{linorder state}$

assumes *finite*: *finite* (*fst*  $\psi$ )

and *snd*  $\psi = \{\}$

shows  $(\exists \psi'. (\text{inference}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable } (\text{fst } \psi)$

using *assms inference-completeness inference-soundness* by *metis*

## 12.4 Lemma about the simplified state

abbreviation *simplified*  $\psi \equiv (\text{no-step simplify } \psi)$

lemma *simplified-count*:

assumes *simp*: *simplified*  $\psi$  and  $\chi: \chi \in \psi$

shows *count*  $\chi$   $L \leq 1$

proof –

{

let  $? \chi' = \chi - \{\#L, L\# \}$

assume *count*  $\chi$   $L \geq 2$

then have *f1*: *count*  $(\chi - \{\#L, L\# \} + \{\#L, L\# \})$   $L = \text{count } \chi$   $L$

by *simp*

then have  $L \in \# \chi - \{\#L\# \}$

by *simp*

then have  $\chi'$ :  $? \chi' + \{\#L\# \} + \{\#L\# \} = \chi$

using *f1* by (*metis* (*no-types*) *diff-diff-add diff-single-eq-union union-assoc union-single-eq-member*)

have  $\exists \psi'. \text{simplify } \psi \psi'$

by (*metis* (*no-types*, *hide-lams*)  $\chi$   $\chi'$  *add.commute factoring-imp-simplify union-assoc*)

then have *False* using *simp* by *auto*

}

then show *?thesis* by *arith*

qed

lemma *simplified-no-both*:

assumes *simp*: *simplified*  $\psi$  and  $\chi: \chi \in \psi$

shows  $\neg (L \in \# \chi \wedge \neg L \in \# \chi)$

proof (*rule ccontr*)

assume  $\neg \neg (L \in \# \chi \wedge \neg L \in \# \chi)$

then have  $L \in \# \chi \wedge \neg L \in \# \chi$  by *metis*

then obtain  $\chi'$  where  $\chi = \chi' + \{\# \text{Pos } (\text{atm-of } L)\# \} + \{\# \text{Neg } (\text{atm-of } L)\# \}$

by (*metis* *Neg-atm-of-iff Pos-atm-of-iff diff-union-swap insert-DiffM2 uminus-Neg uminus-Pos*)

then show *False* using  $\chi$  *simp tautology-deletion* by *fastforce*

qed

lemma *simplified-not-tautology*:

assumes *simplified*  $\{\psi\}$

shows  $\sim \text{tautology } \psi$

```

proof (rule ccontr)
  assume  $\sim$  ?thesis
  then obtain  $p$  where  $Pos\ p \in \# \psi \wedge Neg\ p \in \# \psi$  using tautology-decomp by metis
  then obtain  $\chi$  where  $\psi = \chi + \{\#Pos\ p\} + \{\#Neg\ p\}$ 
    by (metis insert-noteq-member literal.distinct(1) multi-member-split)
  then have  $\sim$  simplified  $\{\psi\}$  by (auto intro: tautology-deletion)
  then show False using assms by auto
qed

```

```

lemma simplified-remove:
  assumes simplified  $\{\psi\}$ 
  shows simplified  $\{\psi - \{\#l\#\}\}$ 
proof (rule ccontr)
  assume  $ns: \neg$  simplified  $\{\psi - \{\#l\#\}\}$ 
  {
    assume  $\neg l \in \# \psi$ 
    then have  $\psi - \{\#l\#\} = \psi$  by simp
    then have False using ns assms by auto
  }
  moreover {
    assume  $l\psi: l \in \# \psi$ 
    have  $A: \bigwedge A. A \in \{\psi - \{\#l\#\}\} \longleftrightarrow A + \{\#l\#\} \in \{\psi\}$  by (auto simp add:  $l\psi$ )
    obtain  $l'$  where  $l':$  simplify  $\{\psi - \{\#l\#\}\}$   $l'$  using ns by metis
    then have  $\exists l'. \text{simplify } \{\psi\} \ l'$ 
    proof (induction rule: simplify.induct)
      case (tautology-deletion  $A\ P$ )
      have  $\{\#Neg\ P\} + (\{\#Pos\ P\} + (A + \{\#l\#\})) \in \{\psi\}$ 
        by (metis (no-types)  $A$  add.commute tautology-deletion.hyps union-lcomm)
      then show ?thesis
        by (metis simplify.tautology-deletion[of  $A + \{\#l\#\}$   $P\ \{\psi\}$ ] add.commute)
    next
      case (condensation  $A\ L$ )
      have  $A + \{\#L\#\} + \{\#L\#\} + \{\#l\#\} \in \{\psi\}$ 
        using  $A$  condensation.hyps by blast
      then have  $\{\#L, L\#\} + (A + \{\#l\#\}) \in \{\psi\}$ 
        by (metis (no-types) union-assoc union-commute)
      then show ?case
        using factoring-imp-simplify by blast
    next
      case (subsumption  $A\ B$ )
      then show ?case by blast
    qed
  }
  then have False using assms(1) by blast
ultimately show False by auto
qed

```

```

lemma in-simplified-simplified:
  assumes simp: simplified  $\psi$  and incl:  $\psi' \subseteq \psi$ 
  shows simplified  $\psi'$ 
proof (rule ccontr)
  assume  $\neg$  ?thesis
  then obtain  $\psi''$  where simplify  $\psi' \ \psi''$  by metis
  then have  $\exists l'. \text{simplify } \psi \ l'$ 

```

```

proof (induction rule: simplify.induct)
  case (tautology-deletion A P)
  then show ?thesis using simplify.tautology-deletion[of A P  $\psi$ ] incl by blast
next
  case (condensation A L)
  then show ?case using simplify.condensation[of A L  $\psi$ ] incl by blast
next
  case (subsumption A B)
  then show ?case using simplify.subsumption[of A  $\psi$  B] incl by auto
qed
then show False using assms(1) by blast
qed

lemma simplified-in:
  assumes simplified  $\psi$ 
  and  $N \in \psi$ 
  shows simplified  $\{N\}$ 
  using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

lemma subsumes-imp-formula:
  assumes  $\psi \leq \# \varphi$ 
  shows  $\{\psi\} \models_p \varphi$ 
  unfolding true-clss-clss-def apply auto
  using assms true-clss-mono-leD by blast

lemma simplified-imp-distinct-mset-tauto:
  assumes simp: simplified  $\psi'$ 
  shows distinct-mset-set  $\psi'$  and  $\forall \chi \in \psi'. \neg \text{tautology } \chi$ 
proof -
  show  $\forall \chi \in \psi'. \neg \text{tautology } \chi$ 
  using simp by (auto simp add: simplified-in simplified-not-tautology)

  show distinct-mset-set  $\psi'$ 
  proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $\chi$  where  $\chi \in \psi'$  and  $\neg \text{distinct-mset } \chi$  unfolding distinct-mset-set-def by auto
  then obtain L where count  $\chi$  L  $\geq 2$ 
  unfolding distinct-mset-def by (metis gr-implies-not0 le-antisym less-one not-le simp
    simplified-count)
  then show False by (metis Suc-1  $\langle \chi \in \psi' \rangle$  not-less-eq-eq simp simplified-count)
  qed
qed

lemma simplified-no-more-full1-simplified:
  assumes simplified  $\psi$ 
  shows  $\neg \text{full1 simplify } \psi \psi'$ 
  using assms unfolding full1-def by (meson tranclpD)

```

## 12.5 Resolution and Invariants

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



### 12.5.1 Invariants

**lemma** *resolution-finite*:

**assumes** *resolution*  $\psi$   $\psi'$  **and** *finite* (*fst*  $\psi$ )  
**shows** *finite* (*fst*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *resolution.induct*)  
*(auto simp add: full1-def full-def rtranclp-simplify-preserves-finite*  
*dest: tranclp-into-rtranclp inference-preserves-finite)*

**lemma** *rtranclp-resolution-finite*:

**assumes** *resolution\*\**  $\psi$   $\psi'$  **and** *finite* (*fst*  $\psi$ )  
**shows** *finite* (*fst*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *rtranclp-induct*, *auto simp add: resolution-finite*)

**lemma** *resolution-finite-snd*:

**assumes** *resolution*  $\psi$   $\psi'$  **and** *finite* (*snd*  $\psi$ )  
**shows** *finite* (*snd*  $\psi'$ )  
**using** *assms* **apply** (*induct* rule: *resolution.induct*, *auto simp add: inference-preserves-finite-snd*)  
**using** *inference-preserves-finite-snd* *snd-conv* **by** *metis*

**lemma** *rtranclp-resolution-finite-snd*:

**assumes** *resolution\*\**  $\psi$   $\psi'$  **and** *finite* (*snd*  $\psi$ )  
**shows** *finite* (*snd*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *rtranclp-induct*, *auto simp add: resolution-finite-snd*)

**lemma** *resolution-always-simplified*:

**assumes** *resolution*  $\psi$   $\psi'$   
**shows** *simplified* (*fst*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *resolution.induct*)  
*(auto simp add: full1-def full-def)*

**lemma** *tranclp-resolution-always-simplified*:

**assumes** *tranclp resolution*  $\psi$   $\psi'$   
**shows** *simplified* (*fst*  $\psi'$ )  
**using** *assms* **by** (*induct* rule: *tranclp.induct*, *auto simp add: resolution-always-simplified*)

**lemma** *resolution-atms-of*:

**assumes** *resolution*  $\psi$   $\psi'$  **and** *finite* (*fst*  $\psi$ )  
**shows** *atms-of-m* (*fst*  $\psi'$ )  $\subseteq$  *atms-of-m* (*fst*  $\psi$ )  
**using** *assms* **apply** (*induct* rule: *resolution.induct*)  
**apply** (*simp add: rtranclp-simplify-atms-of-m tranclp-into-rtranclp full1-def* )  
**by** (*metis* (*no-types*, *lifting*) *contra-subsetD* *fst-conv* *full-def*  
*inference-preserves-atms-of-m rtranclp-simplify-atms-of-m subsetI*)

**lemma** *rtranclp-resolution-atms-of*:

**assumes** *resolution\*\**  $\psi$   $\psi'$  **and** *finite* (*fst*  $\psi$ )  
**shows** *atms-of-m* (*fst*  $\psi'$ )  $\subseteq$  *atms-of-m* (*fst*  $\psi$ )  
**using** *assms* **apply** (*induct* rule: *rtranclp-induct*)  
**using** *resolution-atms-of* *rtranclp-resolution-finite* **by** *blast+*

**lemma** *resolution-include*:

**assumes** *res: resolution*  $\psi$   $\psi'$  **and** *finite: finite* (*fst*  $\psi$ )  
**shows** *fst*  $\psi' \subseteq$  *build-all-simple-cls* (*atms-of-m* (*fst*  $\psi$ ))

**proof** –

**have** *finite'*: *finite* (*fst*  $\psi'$ ) **using** *local.finite* *res* *resolution-finite* **by** *blast*  
**have** *simplified* (*fst*  $\psi'$ ) **using** *res* *finite'* *resolution-always-simplified* **by** *blast*

**then have**  $\text{fst } \psi' \subseteq \text{build-all-simple-clss } (\text{atms-of-m } (\text{fst } \psi'))$   
**using** *simplified-in-build-all finite' simplified-imp-distinct-mset-tauto*[of  $\text{fst } \psi'$ ] **by auto**  
**moreover have**  $\text{atms-of-m } (\text{fst } \psi') \subseteq \text{atms-of-m } (\text{fst } \psi)$   
**using** *res finite resolution-atms-of*[of  $\psi \psi'$ ] **by auto**  
**ultimately show** ?thesis **by** (*meson atms-of-m-finite local.finite order.trans rev-finite-subset build-all-simple-clss-mono*)  
**qed**

**lemma** *rtrancpl-resolution-include:*

**assumes** *res: trancpl resolution  $\psi \psi'$  and finite: finite (fst  $\psi$ )*  
**shows**  $\text{fst } \psi' \subseteq \text{build-all-simple-clss } (\text{atms-of-m } (\text{fst } \psi))$   
**using** *assms apply (induct rule: trancpl.induct)*  
**apply** (*simp add: resolution-include*)  
**by** (*meson atms-of-m-finite build-all-simple-clss-finite build-all-simple-clss-mono finite-subset resolution-include rtrancpl-resolution-atms-of set-rev-mp subsetI trancpl-into-rtrancpl*)

**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**  $\text{vars} \subseteq \text{vars}'$   
**shows** *already-used-all-simple a vars  $\implies$  already-used-all-simple a vars'*  
**using** *assms by fast*

**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**  $\text{atms-of-m } (\text{fst } S) \subseteq \text{vars}$   
**shows** *already-used-all-simple (snd (fst S  $\cup$  {fst S'}, snd S')) vars*  
**using** *assms*

**proof** (*induct rule: inference-clause.induct*)

**case** (*factoring L C N already-used*)

**then show** ?case **by** (*simp add: simplified-in factoring-imp-simplify*)

**next**

**case** (*resolution P C N D already-used*) **note**  $H = \text{this}$

**show** ?case **apply** *clarify*

**proof** –

**fix**  $A B v$

**assume**  $(A, B) \in \text{snd } (\text{fst } (N, \text{already-used}))$

$\cup \{\text{fst } (C + D, \text{already-used} \cup \{(\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)\}),$   
 $\text{snd } (C + D, \text{already-used} \cup \{(\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)\})\}$

**then have**  $(A, B) \in \text{already-used} \vee (A, B) = (\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)$  **by auto**

**moreover** {

**assume**  $(A, B) \in \text{already-used}$

**then have**  $\text{simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars}$

**using**  $H(4)$  **by auto**

}

**moreover** {

**assume**  $\text{eq: } (A, B) = (\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)$

**then have**  $\text{simplified } \{A\}$  **using** *simplified-in H(1,5)* **by auto**

**moreover have**  $\text{simplified } \{B\}$  **using** *eq simplified-in H(2,5)* **by auto**

**moreover have**  $\text{atms-of } A \subseteq \text{atms-of-m } N$

```

      using eq H(1) atms-of-atms-of-m-mono[of A N] by auto
    moreover have atms-of B  $\subseteq$  atms-of-m N
      using eq H(2) atms-of-atms-of-m-mono[of B N] by auto
    ultimately have simplified {A}  $\wedge$  simplified {B}  $\wedge$  atms-of A  $\subseteq$  vars  $\wedge$  atms-of B  $\subseteq$  vars
      using H(6) by auto
  }
  ultimately show simplified {A}  $\wedge$  simplified {B}  $\wedge$  atms-of A  $\subseteq$  vars  $\wedge$  atms-of B  $\subseteq$  vars
    by fast
qed
qed

```

```

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-m (fst S)  $\subseteq$  vars
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: inference.induct)
  case (inference-step S clause already-used)
  then show ?case
    using inference-clause-preserves-already-used-all-simple[of S (clause, already-used) vars]
    by auto
qed

```

```

lemma already-used-all-simple-inv:
  assumes resolution S S'
  and already-used-all-simple (snd S) vars
  and atms-of-m (fst S)  $\subseteq$  vars
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: resolution.induct)
  case (full1-simp N N')
  then show ?case by simp
next
  case (inferring N already-used N' already-used' N'')
  then show already-used-all-simple (snd (N'', already-used')) vars
    using inference-preserves-already-used-all-simple[of (N, already-used)] by simp
qed

```

```

lemma rtrancpl-already-used-all-simple-inv:
  assumes resolution** S S'
  and already-used-all-simple (snd S) vars
  and atms-of-m (fst S)  $\subseteq$  vars
  and finite (fst S)
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: rtrancpl-induct)
  case base
  then show ?case by simp
next
  case (step S' S'')
  note infstar = this(1) and IH = this(3) and res = this(2) and
    already = this(4) and atms = this(5) and finite = this(6)
  have already-used-all-simple (snd S') vars using IH already atms finite by simp
  moreover have atms-of-m (fst S')  $\subseteq$  atms-of-m (fst S)

```

by (simp add: infstar local.finite rtranclp-resolution-atms-of)  
 then have  $\text{atms-of-}m \text{ (fst } S') \subseteq \text{vars}$  using *atms* by auto  
 ultimately show ?case  
 using *already-used-all-simple-inv*[OF *res*] by simp  
 qed

**lemma** *inference-clause-simplified-already-used-subset*:  
 assumes *inference-clause*  $S \ S'$   
 and *simplified* (fst  $S$ )  
 shows  $\text{snd } S \subset \text{snd } S'$   
 using *assms* apply (induct rule: *inference-clause.induct*, auto)  
 using *factoring-imp-simplify* by blast

**lemma** *inference-simplified-already-used-subset*:  
 assumes *inference*  $S \ S'$   
 and *simplified* (fst  $S$ )  
 shows  $\text{snd } S \subset \text{snd } S'$   
 using *assms* apply (induct rule: *inference.induct*)  
 by (metis *inference-clause-simplified-already-used-subset* *snd-conv*)

**lemma** *resolution-simplified-already-used-subset*:  
 assumes *resolution*  $S \ S'$   
 and *simplified* (fst  $S$ )  
 shows  $\text{snd } S \subset \text{snd } S'$   
 using *assms* apply (induct rule: *resolution.induct*, *simp-all* add: *full1-def*)  
 apply (meson *tranclpD*)  
 by (metis *inference-simplified-already-used-subset* *fst-conv* *snd-conv*)

**lemma** *tranclp-resolution-simplified-already-used-subset*:  
 assumes *tranclp* *resolution*  $S \ S'$   
 and *simplified* (fst  $S$ )  
 shows  $\text{snd } S \subset \text{snd } S'$   
 using *assms* apply (induct rule: *tranclp.induct*)  
 using *resolution-simplified-already-used-subset* apply metis  
 by (meson *tranclp-resolution-always-simplified* *resolution-simplified-already-used-subset* *less-trans*)

**abbreviation** *already-used-top vars*  $\equiv \text{build-all-simple-clss vars} \times \text{build-all-simple-clss vars}$

**lemma** *already-used-all-simple-in-already-used-top*:  
 assumes *already-used-all-simple*  $s \ \text{vars}$  and *finite vars*  
 shows  $s \subseteq \text{already-used-top vars}$   
**proof**  
 fix  $x$   
 assume  $x-s: x \in s$   
 obtain  $A \ B$  where  $x: x = (A, B)$  by (case-tac  $x$ , auto)  
 then have *simplified*  $\{A\}$  and  $\text{atms-of } A \subseteq \text{vars}$  using *assms*(1)  $x-s$  by *fastforce* +  
 then have  $A: A \in \text{build-all-simple-clss vars}$   
 using *build-all-simple-clss-mono*[of *vars* *atms-of A*]  $x$  *assms*(2)  
*simplified-imp-distinct-mset-tauto*[of  $\{A\}$ ]  
*distinct-mset-not-tautology-implies-in-build-all-simple-clss* by *fast*  
 moreover have *simplified*  $\{B\}$  and  $\text{atms-of } B \subseteq \text{vars}$  using *assms*(1)  $x-s \ x$  by *fast* +  
 then have  $B: B \in \text{build-all-simple-clss vars}$   
 using *simplified-imp-distinct-mset-tauto*[of  $\{B\}$ ]  
*distinct-mset-not-tautology-implies-in-build-all-simple-clss*

$\text{build-all-simple-clss-mono}[\text{of vars atms-of } B] \ x \ \text{assms}(2) \ \mathbf{by} \ \text{fast}$   
**ultimately show**  $x \in \text{build-all-simple-clss vars} \times \text{build-all-simple-clss vars}$   
 unfolding  $x$  **by** *auto*  
**qed**

**lemma** *already-used-top-finite*:  
 assumes *finite vars*  
 shows *finite (already-used-top vars)*  
 using *build-all-simple-clss-finite assms* **by** *auto*

**lemma** *already-used-top-increasing*:  
 assumes  $\text{var} \subseteq \text{var}'$  **and** *finite var'*  
 shows *already-used-top var  $\subseteq$  already-used-top var'*  
 using *assms build-all-simple-clss-mono* **by** *auto*

**lemma** *already-used-all-simple-finite*:  
 fixes  $s :: ('a::\text{linorder literal multiset} \times 'a \text{ literal multiset}) \text{ set}$  **and**  $\text{vars} :: 'a \text{ set}$   
 assumes *already-used-all-simple s vars* **and** *finite vars*  
 shows *finite s*  
 using *assms already-used-all-simple-in-already-used-top[OF assms(1)]*  
*rev-finite-subset[OF already-used-top-finite[of vars]]* **by** *auto*

**abbreviation** *card-simple vars  $\psi \equiv \text{card (already-used-top vars} - \psi)$*

**lemma** *resolution-card-simple-decreasing*:  
 assumes *res: resolution  $\psi \ \psi'$*   
**and** *a-u-s: already-used-all-simple (snd  $\psi$ ) 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-m (fst  $\psi$ )  $\subseteq$  vars*  
 shows *card-simple vars (snd  $\psi'$ )  $<$  card-simple vars (snd  $\psi$ )*

**proof** –  
 let  $?vars = \text{vars}$   
 let  $?top = \text{build-all-simple-clss } ?vars \times \text{build-all-simple-clss } ?vars$   
 have 1: *card-simple vars (snd  $\psi$ ) = card ?top – card (snd  $\psi$ )*  
 using *card-Diff-subset finite-snd already-used-all-simple-in-already-used-top[OF a-u-s]*  
*finite-v* **by** *metis*  
 have *a-u-s': already-used-all-simple (snd  $\psi'$ ) vars*  
 using *already-used-all-simple-inv res a-u-s assms(7)* **by** *blast*  
 have *f: finite (snd  $\psi'$ )* **using** *already-used-all-simple-finite a-u-s' finite-v* **by** *auto*  
 have 2: *card-simple vars (snd  $\psi'$ ) = card ?top – card (snd  $\psi'$ )*  
 using *card-Diff-subset[OF f] already-used-all-simple-in-already-used-top[OF a-u-s' finite-v]*  
**by** *auto*  
 have *card (already-used-top vars)  $\geq$  card (snd  $\psi'$ )*  
 using *already-used-all-simple-in-already-used-top[OF a-u-s' finite-v]*  
*card-mono[of already-used-top vars snd  $\psi'$ ] already-used-top-finite[OF finite-v]* **by** *metis*  
**then show** *?thesis*  
 using *psubset-card-mono[OF f resolution-simplified-already-used-subset[OF res simp]]*  
 unfolding 1 2 **by** *linarith*  
**qed**

**lemma** *trancp-resolution-card-simple-decreasing*:

**assumes** *trancpl resolution*  $\psi \psi'$  **and** *finite-fst*: *finite* (*fst*  $\psi$ )  
**and** *already-used-all-simple* (*snd*  $\psi$ ) *vars*  
**and** *atms-of-m* (*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*  $\psi$ )  
**using** *assms*  
**proof** (*induct rule*: *trancpl.induct*)  
**case** (*r-into-trancpl*  $\psi \psi'$ )  
**then show** ?*case* **by** (*simp add*: *resolution-card-simple-decreasing*)  
**next**  
**case** (*trancpl-into-trancpl*  $\psi \psi' \psi''$ ) **note** *res* = *this*(1) **and** *res'* = *this*(3) **and** *a-u-s* = *this*(5) **and**  
*atms* = *this*(6) **and** *f-v* = *this*(7) **and** *f-fst* = *this*(4) **and** *H* = *this*  
**then have** *card-simple vars* (*snd*  $\psi'$ )  $<$  *card-simple vars* (*snd*  $\psi$ ) **by** *auto*  
**moreover have** *a-u-s'*: *already-used-all-simple* (*snd*  $\psi'$ ) *vars*  
**using** *rtrancpl-already-used-all-simple-inv*[*OF* *trancpl-into-rtrancpl*[*OF* *res*] *a-u-s* *atms* *f-fst*] .  
**have** *finite* (*fst*  $\psi'$ )  
**by** (*meson build-all-simple-clss-finite rev-finite-subset rtrancpl-resolution-include*  
*trancpl-into-trancpl.hyps*(1) *trancpl-into-trancpl.prem*s(1))  
**moreover have** *finite* (*snd*  $\psi'$ ) **using** *already-used-all-simple-finite*[*OF* *a-u-s'* *f-v*] .  
**moreover have** *simplified* (*fst*  $\psi'$ ) **using** *res* *trancpl-resolution-always-simplified* **by** *blast*  
**moreover have** *atms-of-m* (*fst*  $\psi'$ )  $\subseteq$  *vars*  
**by** (*meson* *atms* *f-fst* *order.trans* *res* *rtrancpl-resolution-atms-of* *trancpl-into-rtrancpl*)  
**ultimately show** ?*case*  
**using** *resolution-card-simple-decreasing*[*OF* *res'* *a-u-s'* *f-v*] *f-v*  
*less-trans*[*of* *card-simple vars* (*snd*  $\psi''$ ) *card-simple vars* (*snd*  $\psi'$ )  
*card-simple vars* (*snd*  $\psi$ )]  
**by** *blast*  
**qed**

**lemma** *trancpl-resolution-card-simple-decreasing-2*:  
**assumes** *trancpl resolution*  $\psi \psi'$   
**and** *finite-fst*: *finite* (*fst*  $\psi$ )  
**and** *empty-snd*: *snd*  $\psi$  = {}  
**and** *simplified* (*fst*  $\psi$ )  
**shows** *card-simple* (*atms-of-m* (*fst*  $\psi$ )) (*snd*  $\psi'$ )  $<$  *card-simple* (*atms-of-m* (*fst*  $\psi$ )) (*snd*  $\psi$ )  
**proof** –  
**let** ?*vars* = (*atms-of-m* (*fst*  $\psi$ ))  
**have** *already-used-all-simple* (*snd*  $\psi$ ) ?*vars* **unfolding** *empty-snd* **by** *auto*  
**moreover have** *atms-of-m* (*fst*  $\psi$ )  $\subseteq$  ?*vars* **by** *auto*  
**moreover have** *finite-v*: *finite* ?*vars* **using** *finite-fst* **by** *auto*  
**moreover have** *finite-snd*: *finite* (*snd*  $\psi$ ) **unfolding** *empty-snd* **by** *auto*  
**ultimately show** ?*thesis*  
**using** *assms*(1,2,4) *trancpl-resolution-card-simple-decreasing*[*of*  $\psi \psi'$ ] **by** *presburger*  
**qed**

## 12.5.2 well-foundedness if the relation

**lemma** *wf-simplified-resolution*:  
**assumes** *f-vars*: *finite vars*  
**shows** *wf* {(*y* :: '*v*': *linorder state*, *x*). (*atms-of-m* (*fst* *x*)  $\subseteq$  *vars*  $\wedge$  *simplified* (*fst* *x*)  
 $\wedge$  *finite* (*snd* *x*)  $\wedge$  *finite* (*fst* *x*)  $\wedge$  *already-used-all-simple* (*snd* *x*) *vars*)  $\wedge$  *resolution* *x y*}  
**proof** –  
{

```

fix a b :: 'v::linorder state
assume (b, a) ∈ {(y, x). (atms-of-m (fst x) ⊆ vars ∧ simplified (fst x) ∧ finite (snd x)
  ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
then have
  atms-of-m (fst a) ⊆ vars and
  simp: simplified (fst a) and
  finite (snd a) and
  finite (fst a) and
  a-u-v: already-used-all-simple (snd a) vars and
  res: resolution a b by auto
have finite (already-used-top vars) using f-vars already-used-top-finite by blast
moreover have already-used-top vars ⊆ already-used-top vars by auto
moreover have snd b ⊆ already-used-top vars
  using already-used-all-simple-in-already-used-top[of snd b vars]
  a-u-v already-used-all-simple-inv[OF res] (finite (fst a)) (atms-of-m (fst a) ⊆ vars) f-vars
  by presburger
moreover have snd a ⊆ snd b using resolution-simplified-already-used-subset[OF res simp] .
ultimately have finite (already-used-top vars) ∧ already-used-top vars ⊆ already-used-top vars
  ∧ snd b ⊆ already-used-top vars ∧ snd a ⊆ snd b by metis
}
then show ?thesis using wf-bounded-set[of {(y:: 'v:: linorder state, x).
  (atms-of-m (fst x) ⊆ vars
  ∧ simplified (fst x) ∧ finite (snd x) ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars)
  ∧ resolution x y} λ-. already-used-top vars snd] by auto
qed

```

```

lemma wf-simplified-resolution':
assumes f-vars: finite vars
shows wf {(y:: 'v:: linorder state, x). (atms-of-m (fst x) ⊆ vars ∧ ¬simplified (fst x)
  ∧ finite (snd x) ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
unfolding wf-def
apply (simp add: resolution-always-simplified)
by (metis (mono-tags, hide-lams) fst-conv resolution-always-simplified)

```

```

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

```

```

proof –
have Domain ?R Int Range ?S = {} using resolution-always-simplified by auto blast
then show wf (?R ∪ ?S)
  using wf-simplified-resolution[OF f-vars] wf-simplified-resolution'[OF f-vars] wf-Un[of ?R ?S]
  by fast
qed

```

```

lemma rtrancp-simplify-already-used-inv:
assumes simplify** S S'
and already-used-inv (S, N)
shows already-used-inv (S', N)
using assms apply induction
using simplify-preserves-already-used-inv by fast+

```

```

lemma full1-simplify-already-used-inv:

```

```

assumes full1 simplify S S'
and already-used-inv (S, N)
shows already-used-inv (S', N)
using assms tranclp-into-rtranclp[of simplify S S'] rtranclp-simplify-already-used-inv
unfolding full1-def by fast

lemma full-simplify-already-used-inv:
  assumes full simplify S S'
  and already-used-inv (S, N)
  shows already-used-inv (S', N)
  using assms rtranclp-simplify-already-used-inv unfolding full-def by fast
lemma resolution-already-used-inv:
  assumes resolution S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
proof induction
  case (full1-simp N N' already-used)
  then show ?case using full1-simplify-already-used-inv by fast
next
  case (inferring N already-used N' already-used' N'') note inf = this(1) and full = this(3) and
    a-u-v = this(4)
  then show ?case
    using inference-preserves-already-used-inv[OF inf a-u-v] full-simplify-already-used-inv full
    by fast
qed

lemma rtranclp-resolution-already-used-inv:
  assumes resolution** S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms apply induction
  using resolution-already-used-inv by fast+

lemma rtanclp-simplify-preserves-unsat:
  assumes simplify**  $\psi$   $\psi'$ 
  shows satisfiable  $\psi' \longrightarrow$  satisfiable  $\psi$ 
  using assms apply induction
  using simplify-clause-preserves-sat by blast+

lemma full1-simplify-preserves-unsat:
  assumes full1 simplify  $\psi$   $\psi'$ 
  shows satisfiable  $\psi' \longrightarrow$  satisfiable  $\psi$ 
  using assms rtanclp-simplify-preserves-unsat[of  $\psi$   $\psi'$ ] tranclp-into-rtranclp
  unfolding full1-def by metis

lemma full-simplify-preserves-unsat:
  assumes full simplify  $\psi$   $\psi'$ 
  shows satisfiable  $\psi' \longrightarrow$  satisfiable  $\psi$ 
  using assms rtanclp-simplify-preserves-unsat[of  $\psi$   $\psi'$ ] unfolding full-def by metis

lemma resolution-preserves-unsat:
  assumes resolution  $\psi$   $\psi'$ 
  shows satisfiable (fst  $\psi'$ )  $\longrightarrow$  satisfiable (fst  $\psi$ )
  using assms apply (induct rule: resolution.induct)

```



```

using full1-simplify-preserves-unsat apply (metis fst-conv)
using full-simplify-preserves-unsat simplify-preserves-unsat by fastforce

lemma rtrancpl-resolution-preserves-unsat:
  assumes resolution**  $\psi \ \psi'$ 
  shows satisfiable (fst  $\psi'$ )  $\longrightarrow$  satisfiable (fst  $\psi$ )
  using assms apply induction
  using resolution-preserves-unsat by fast+

lemma rtrancpl-simplify-preserve-partial-tree:
  assumes simplify**  $N \ N'$ 
  and partial-interps  $t \ I \ N$ 
  shows partial-interps  $t \ I \ N'$ 
  using assms apply (induction, simp)
  using simplify-preserve-partial-tree by metis

lemma full1-simplify-preserve-partial-tree:
  assumes full1 simplify  $N \ N'$ 
  and partial-interps  $t \ I \ N$ 
  shows partial-interps  $t \ I \ N'$ 
  using assms rtrancpl-simplify-preserve-partial-tree[of  $N \ N' \ t \ I$ ] trancpl-into-rtrancpl
  unfolding full1-def by fast

lemma full-simplify-preserve-partial-tree:
  assumes full simplify  $N \ N'$ 
  and partial-interps  $t \ I \ N$ 
  shows partial-interps  $t \ I \ N'$ 
  using assms rtrancpl-simplify-preserve-partial-tree[of  $N \ N' \ t \ I$ ] trancpl-into-rtrancpl
  unfolding full-def by fast

lemma resolution-preserve-partial-tree:
  assumes resolution  $S \ S'$ 
  and partial-interps  $t \ I \ (\text{fst } S)$ 
  shows partial-interps  $t \ I \ (\text{fst } S')$ 
  using assms apply induction
  using full1-simplify-preserve-partial-tree fst-conv apply metis
  using full-simplify-preserve-partial-tree inference-preserve-partial-tree by fastforce

lemma rtrancpl-resolution-preserve-partial-tree:
  assumes resolution**  $S \ S'$ 
  and partial-interps  $t \ I \ (\text{fst } S)$ 
  shows partial-interps  $t \ I \ (\text{fst } S')$ 
  using assms apply induction
  using resolution-preserve-partial-tree by fast+
  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  $P \ n$ 
  using assms apply (induct rule: nat-less-induct)
  by (case-tac  $n$ ) auto

lemma wf-always-more-step-False:
  assumes wf  $R$ 

```

**shows**  $(\forall x. \exists z. (z, x) \in R) \implies \text{False}$   
**using** *assms* **unfolding** *wf-def* **by** (*meson* *Domain.DomainI* *assms* *wfE-min*)

**lemma** *finite-finite-mset-element-of-mset[simp]*:

**assumes** *finite N*  
**shows** *finite*  $\{f \varphi L \mid \varphi L. \varphi \in N \wedge L \in \# \varphi \wedge P \varphi L\}$   
**using** *assms*

**proof** (*induction N* *rule: finite-induct*)

**case** *empty*  
**show** *?case* **by** *auto*

**next**

**case** (*insert x N*) **note** *finite = this(1)* **and** *IH = this(3)*  
**have**  $\{f \varphi L \mid \varphi L. (\varphi = x \vee \varphi \in N) \wedge L \in \# \varphi \wedge P \varphi L\} \subseteq \{f x L \mid L. L \in \# x \wedge P x L\}$   
 $\cup \{f \varphi L \mid \varphi L. \varphi \in N \wedge L \in \# \varphi \wedge P \varphi L\}$  **by** *auto*  
**moreover** **have** *finite*  $\{f x L \mid L. L \in \# x\}$  **by** *auto*  
**ultimately show** *?case* **using** *IH* *finite-subset* **by** *fastforce*

**qed**

**value** *card*

**value** *filter-mset*

**value**  $\{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \#\}$

**value**  $(\lambda \varphi. \text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \#\})$

**syntax**

*-comprehension1'-mset*  $:: 'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset}$   
 $((\{\# \cdot / \cdot - : \text{setof } \cdot \#\}))$

**translations**

$\{\# e. x : \text{setof } M \#\} == \text{CONST set-mset } (\text{CONST image-mset } (\%x. e) M)$

**value**  $\{\# a. a : \text{setof } \{\# 1, 1, 2 :: \text{int}\} \#\} = \{1, 2\}$

**definition** *sum-count-ge-2*  $:: 'a \text{ multiset set} \Rightarrow \text{nat } (\Xi)$  **where**

*sum-count-ge-2*  $\equiv \text{folding.F } (\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \#\})) 0$

**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*  $(\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \#\})) 0 = \text{sum-count-ge-2}$

**proof** –

**show** *folding*  $(\lambda \varphi. \text{op} + (\text{msetsum } (\text{image-mset } (\text{count } \varphi) \{\# L : \# \varphi. 2 \leq \text{count } \varphi L \#\})))$   
**by** *standard auto*

**then interpret** *sum-count-ge-2*:

*folding*  $(\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \#\})) 0 .$

**show** *folding.F*  $(\lambda \varphi. \text{op} + (\text{msetsum } (\text{image-mset } (\text{count } \varphi) \{\# L : \# \varphi. 2 \leq \text{count } \varphi L \#\}))) 0$   
 $= \text{sum-count-ge-2}$  **by** (*auto simp add: sum-count-ge-2-def*)

**qed**

**lemma** *finite-incl-le-setsum*:

*finite*  $(B :: 'a \text{ multiset set}) \implies A \subseteq B \implies \Xi A \leq \Xi B$

**proof** (*induction arbitrary:A* *rule: finite-induct*)

**case** *empty*

**then show** *?case* **by** *simp*

**next**

**case** (*insert a F*) **note** *finite = this(1)* **and** *aF = this(2)* **and** *IH = this(3)* **and** *AF = this(4)*

```

show ?case
proof (cases a ∈ A)
  assume a ∉ A
  then have A ⊆ F using AF by auto
  then show ?case using IH[of A] by (simp add: aF local.finite)
next
  assume aA: a ∈ A
  then have A - {a} ⊆ F using AF by auto
  then have ∃ (A - {a}) ≤ ∃ F using IH by blast
  then show ?case
    proof -
      obtain nn :: nat ⇒ nat ⇒ nat where
        ∀ x0 x1. (∃ v2. x0 = x1 + v2) = (x0 = x1 + nn x0 x1)
      by moura
      then have ∃ F = ∃ (A - {a}) + nn (∃ F) (∃ (A - {a}))
        using Nat.le-iff-add ⟨∃ (A - {a}) ≤ ∃ F⟩ by presburger
      then show ?thesis
        by (metis (no-types) Nat.le-iff-add aA aF add.assoc finite.insertI finite-subset
            insert.premis local.finite sum-count-ge-2.insert sum-count-ge-2.remove)
    qed
  qed
qed

lemma mset-condensation1:
  {# La :# A + {#L#}. 2 ≤ count (A + {#L#}) La#} = {# La :# A. La ≠ L ∧ 2 ≤ count A
  La#}
  # ∪ (if count A L ≥ 1 then replicate-mset (count A L + 1) L else {#})
  by (auto intro: multiset-eqI)

lemma mset-condensation2:
  {# La :# A + {#L#} + {#L#}. 2 ≤ count (A + {#L#} + {#L#}) La#} = {# La :# A. La ≠
  L ∧
  2 ≤ count A La#} # ∪ (replicate-mset (count A L + 2) L)
  by (auto intro: multiset-eqI)

lemma msetsum-disjoint:
  assumes A # ∩ B = {#}
  shows (∑ La ∈ #A # ∪ B. f La) =
    (∑ La ∈ #A. f La) + (∑ La ∈ #B. f La)
  by (metis assms diff-zero empty-sup image-mset-union msetsum.union multiset-inter-commute
    multiset-union-diff-commute sup-subset-mset-def zero-diff)

lemma msetsum-linear[simp]:
  fixes C D :: 'a ⇒ 'b :: {comm-monoid-add}
  shows (∑ x ∈ #A. C x + D x) = (∑ x ∈ #A. C x) + (∑ x ∈ #A. D x)
  by (induction A) (auto simp: ac-simps)

lemma msetsum-if-eq[simp]: (∑ x ∈ #A. if L = x then 1 else 0) = count A L
  by (induction A) auto

lemma filter-equality-in-mset:
  filter-mset (op = L) A = replicate-mset (count A L) L
  by (auto simp: multiset-eq-iff)

```

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

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

**by** (*auto simp: multiset-eq-iff*)

**lemma** *simplify-finite-measure-decrease*:

$\text{simplify } N \ N' \implies \text{finite } N \implies \text{card } N' + \Xi N' < \text{card } N + \Xi N$

**proof** (*induction rule: simplify.induct*)

**case** (*tautology-deletion*  $A \ P$ ) **note**  $an = \text{this}(1)$  **and**  $fin = \text{this}(2)$

**let**  $?N' = N - \{A + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}\}$

**have**  $\text{card } ?N' < \text{card } N$

**by** (*meson card-Diff1-less tautology-deletion.hyps tautology-deletion.prem*s)

**moreover** **have**  $?N' \subseteq N$  **by** *auto*

**then** **have**  $\text{sum-count-ge-2 } ?N' \leq \text{sum-count-ge-2 } N$  **using** *finite-incl-le-setsum[OF fin]* **by** *blast*

**ultimately** **show**  $?case$  **by** *linarith*

**next**

**case** (*condensation*  $A \ L$ ) **note**  $AN = \text{this}(1)$  **and**  $fin = \text{this}(2)$

**let**  $?C' = A + \{\#L\# \}$

**let**  $?C = A + \{\#L\# \} + \{\#L\# \}$

**let**  $?N' = N - \{?C\} \cup \{?C'\}$

**have**  $\text{card } ?N' \leq \text{card } N$

**using**  $AN$  **by** (*metis (no-types, lifting) Diff-subset Un-empty-right Un-insert-right card.remove card-insert-if card-mono fin finite-Diff order-refl*)

**moreover** **have**  $\Xi \{?C'\} < \Xi \{?C\}$

**proof** –

**have** *mset-decomp*:

$\{\# La \in \# A. (L = La \longrightarrow \text{Suc } 0 \leq \text{count } A \ La) \wedge (L \neq La \longrightarrow 2 \leq \text{count } A \ La)\# \}$

$= \{\# La \in \# A. L \neq La \wedge 2 \leq \text{count } A \ La\# \} +$

$\{\# La \in \# A. L = La \wedge \text{Suc } 0 \leq \text{count } A \ L\# \}$

**by** (*auto simp: multiset-eq-iff ac-simps*)

**have** *mset-decomp2*:  $\{\# La \in \# A. L \neq La \longrightarrow 2 \leq \text{count } A \ La\# \} =$

$\{\# La \in \# A. L \neq La \wedge 2 \leq \text{count } A \ La\# \} + \text{replicate-mset } (\text{count } A \ L) \ L$

**by** (*auto simp: multiset-eq-iff*)

**show** *?thesis*

**by** (*auto simp: mset-decomp mset-decomp2 filter-equality-in-mset ac-simps*)

**qed**

**have**  $\Xi ?N' < \Xi N$

**proof** *cases*

**assume**  $a1: ?C' \in N$

**then** **show** *?thesis*

**proof** –

**have**  $f2: \bigwedge m \ M. \text{insert } (m::'a \text{ literal multiset}) (M - \{m\}) = M \cup \{ \} \vee m \notin M$

**using** *Un-empty-right insert-Diff* **by** *blast*

**have**  $f3: \bigwedge m \ M \ Ma. \text{insert } (m::'a \text{ literal multiset}) M - \text{insert } m \ Ma = M - \text{insert } m \ Ma$

**by** *simp*

**then** **have**  $f4: \bigwedge M \ m. M - \{m::'a \text{ literal multiset}\} = M \cup \{ \} \vee m \in M$

**using** *Diff-insert-absorb Un-empty-right* **by** *fastforce*

**have**  $f5: \text{insert } (A + \{\#L\# \} + \{\#L\# \}) N = N$

**using**  $f3 \ f2$  *Un-empty-right condensation.hyps insert-iff* **by** *fastforce*

**have**  $\bigwedge m \ M. \text{insert } (m::'a \text{ literal multiset}) M = M \cup \{ \} \vee m \notin M$

**using**  $f3 \ f2$  *Un-empty-right add.right-neutral insert-iff* **by** *fastforce*

**then** **have**  $\Xi (N - \{A + \{\#L\# \} + \{\#L\# \}) < \Xi N$

**using**  $f5 \ f4$  **by** (*metis Un-empty-right*  $\Xi \{A + \{\#L\# \}\} < \Xi \{A + \{\#L\# \} + \{\#L\# \}\}$ )

*add.right-neutral add-diff-cancel-left' add-gr-0 diff-less fin finite.emptyI not-le*

*sum-count-ge-2.empty sum-count-ge-2.insert-remove trans-le-add2*)

```

    then show ?thesis
    using f3 f2 a1 by (metis (no-types) Un-empty-right Un-insert-right condensation.hyps
        insert-iff multi-self-add-other-not-self)
  qed
next
  assume ?C'  $\notin$  N
  have mset-decomp:
    {# La  $\in$  # A. (L = La  $\longrightarrow$  Suc 0  $\leq$  count A La)  $\wedge$  (L  $\neq$  La  $\longrightarrow$  2  $\leq$  count A La)#}
    = {# La  $\in$  # A. L  $\neq$  La  $\wedge$  2  $\leq$  count A La#} +
      {# La  $\in$  # A. L = La  $\wedge$  Suc 0  $\leq$  count A L#}
    by (auto simp: multiset-eq-iff ac-simps)
  have mset-decomp2: {# La  $\in$  # A. L  $\neq$  La  $\longrightarrow$  2  $\leq$  count A La#} =
    {# La  $\in$  # A. L  $\neq$  La  $\wedge$  2  $\leq$  count A La#} + replicate-mset (count A L) L
    by (auto simp: multiset-eq-iff)

  show ?thesis
  using  $\langle \exists \{A + \{ \#L\# \} \} < \Xi \{A + \{ \#L\# \} + \{ \#L\# \} \} \rangle$  condensation.hyps fin
    sum-count-ge-2.remove[of - A + {#L#} + {#L#}]  $\langle ?C' \notin N \rangle$ 
    by (auto simp: mset-decomp mset-decomp2 filter-equality-in-mset)
  qed
ultimately show ?case by linarith
next
  case (subsumption A B) note AN = this(1) and AB = this(2) and BN = this(3) and fin = this(4)
  have card (N - {B}) < card N using BN by (meson card-Diff1-less subsumption.prem)
  moreover have  $\Xi (N - \{B\}) \leq \Xi N$ 
    by (simp add: Diff-subset finite-incl-le-setsum subsumption.prem)
  ultimately show ?case by linarith
qed

lemma simplify-terminates:
  wf {(N', N). finite N  $\wedge$  simplify N N'}
  using assms apply (rule wfP-if-measure[of finite simplify  $\lambda N$ . card N +  $\Xi N$ ])
  using simplify-finite-measure-decrease by blast

lemma wf-terminates:
  assumes wf r
  shows  $\exists N'. (N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r)$ 
proof -
  let ?P =  $\lambda N. (\exists N'. (N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r))$ 
  have  $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P y) \longrightarrow ?P x)$ 
  proof clarify
    fix x
    assume H:  $\forall y. (y, x) \in r \longrightarrow ?P y$ 
    { assume  $\exists y. (y, x) \in r$ 
      then obtain y where  $y: (y, x) \in r$  by blast
      then have ?P y using H by blast
      then have ?P x using y by (meson rtrancl.rtrancl-into-rtrancl)
    }
    moreover {
      assume  $\neg(\exists y. (y, x) \in r)$ 
      then have ?P x by auto
    }
  }
  ultimately show ?P x by blast

```

qed  
 moreover have  $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P\ y) \longrightarrow ?P\ x) \longrightarrow \text{All } ?P$   
 using *assms unfolding wf-def* by (rule *allE*)  
 ultimately have  $\text{All } ?P$  by *blast*  
 then show  $?P\ N$  by *blast*  
 qed

lemma *rtrancl-simplify-terminates*:

assumes *fin*: *finite N*  
 shows  $\exists N'. \text{simplify}^{**}\ N\ N' \wedge \text{simplified}\ N'$

proof –

have *H*:  $\{(N', N). \text{finite}\ N \wedge \text{simplify}\ N\ N'\} = \{(N', N). \text{simplify}\ N\ N' \wedge \text{finite}\ N\}$  by *auto*

then have *wf*: *wf*  $\{(N', N). \text{simplify}\ N\ N' \wedge \text{finite}\ N\}$

using *simplify-terminates* by (simp add: *H*)

obtain *N'* where *N'*:  $(N', N) \in \{(b, a). \text{simplify}\ a\ b \wedge \text{finite}\ a\}^*$  and

more:  $(\forall N''. (N'', N') \notin \{(b, a). \text{simplify}\ a\ b \wedge \text{finite}\ a\})$

using *Prop-Resolution.wf-terminates[OF wf, of N]* by *blast*

have *1*:  $\text{simplify}^{**}\ N\ N'$

using *N'* by (induction rule: *rtrancl.induct*) *auto*

then have *finite N'* using *fin rtrancl-simplify-preserves-finite* by *blast*

then have *2*:  $\forall N''. \neg \text{simplify}\ N'\ N''$  using *more* by *auto*

show *?thesis* using *1 2* by *blast*

qed

lemma *finite-simplified-full1-simp*:

assumes *finite N*

shows  $\text{simplified}\ N \vee (\exists N'. \text{full1}\ \text{simplify}\ N\ N')$

using *rtrancl-simplify-terminates[OF assms]* unfolding *full1-def*

by (metis *Nitpick.rtrancl-unfold*)

lemma *finite-simplified-full-simp*:

assumes *finite N*

shows  $\exists N'. \text{full}\ \text{simplify}\ N\ N'$

using *rtrancl-simplify-terminates[OF assms]* unfolding *full-def* by *metis*

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* *tree I* (*fst*  $\psi$ )

and *simplified* (*fst*  $\psi$ )

shows  $\exists (\text{tree}' :: 'v\ \text{sem-tree})\ \psi'. \text{resolution}^{**}\ \psi\ \psi' \wedge \text{partial-interps}\ \text{tree}'\ I\ (\text{fst}\ \psi')$

$\wedge (\text{sem-tree-size}\ \text{tree}' < \text{sem-tree-size}\ \text{tree} \vee \text{sem-tree-size}\ \text{tree} = 0)$

using *assms*

proof (induct arbitrary: *I* rule: *sem-tree-size*)

case (*bigger xs I*) note *IH* = *this(1)* and *finite* = *this(2)* and *a-u-i* = *this(3)* and *part* = *this(4)*

and *simp* = *this(5)*

{ assume *sem-tree-size xs* = 0

then have *?case* using *part* by *blast*

}

moreover {

assume *sn0*: *sem-tree-size xs* > 0

obtain *ag ad v* where *xs*: *xs* = *Node v ag ad* using *sn0* by (case-tac *xs*, *auto*)

```

{
  assume sem-tree-size  $ag = 0 \wedge \text{sem-tree-size } ad = 0$ 
  then have  $ag: ag = \text{Leaf}$  and  $ad: ad = \text{Leaf}$  by (case-tac  $ag$ , auto, case-tac  $ad$ , auto)

  then obtain  $\chi \chi'$  where
     $\chi: \neg I \cup \{\text{Pos } v\} \models \chi$  and
     $\text{tot}\chi: \text{total-over-}m (I \cup \{\text{Pos } v\}) \{\chi\}$  and
     $\chi\psi: \chi \in \text{fst } \psi$  and
     $\chi': \neg I \cup \{\text{Neg } v\} \models \chi'$  and
     $\text{tot}\chi': \text{total-over-}m (I \cup \{\text{Neg } v\}) \{\chi'\}$  and  $\chi'\psi: \chi' \in \text{fst } \psi$ 
    using part unfolding  $xs$  by auto
  have  $\text{Pos}v: \text{Pos } v \notin \chi$  using  $\chi$  unfolding true-cls-def true-lit-def by auto
  have  $\text{Neg}v: \text{Neg } v \notin \chi'$  using  $\chi'$  unfolding true-cls-def true-lit-def by auto
  {
    assume  $\text{Neg}\chi: \neg \text{Neg } v \in \chi$ 
    then have  $\neg I \models \chi$  using  $\chi$   $\text{Pos}v$  unfolding true-cls-def true-lit-def by auto
    moreover have  $\text{total-over-}m I \{\chi\}$ 
      using  $\text{Pos}v \text{Neg}\chi \text{atm-imp-pos-or-neg-lit tot}\chi$  unfolding total-over- $m$ -def total-over-set-def
      by fastforce
    ultimately have partial-interps  $\text{Leaf } I (\text{fst } \psi)$ 
    and sem-tree-size  $\text{Leaf} < \text{sem-tree-size } xs$ 
    and resolution**  $\psi \psi$ 
      unfolding  $xs$  by (auto simp add:  $\chi\psi$ )
  }
  moreover {
    assume  $\text{Pos}\chi: \neg \text{Pos } v \in \chi'$ 
    then have  $I\chi: \neg I \models \chi'$  using  $\chi'$   $\text{Pos}v$  unfolding true-cls-def true-lit-def by auto
    moreover have  $\text{total-over-}m I \{\chi'\}$ 
      using  $\text{Neg}v \text{Pos}\chi \text{atm-imp-pos-or-neg-lit tot}\chi'$ 
      unfolding total-over- $m$ -def total-over-set-def by fastforce
    ultimately have partial-interps  $\text{Leaf } I (\text{fst } \psi)$ 
    and sem-tree-size  $\text{Leaf} < \text{sem-tree-size } xs$ 
    and resolution**  $\psi \psi$  using  $\chi'\psi I\chi$  unfolding  $xs$  by auto
  }
  moreover {
    assume  $\text{neg}: \text{Neg } v \in \chi$  and  $\text{pos}: \text{Pos } v \in \chi'$ 
    have count  $\chi (\text{Neg } v) = 1$ 
      using simplified-count[OF simp  $\chi\psi$ ]  $\text{neg}$  by (metis One-nat-def Suc-le-mono Suc-pred eq-iff le0)
    have count  $\chi' (\text{Pos } v) = 1$ 
      using simplified-count[OF simp  $\chi'\psi$ ]  $\text{pos}$  by (metis One-nat-def Suc-le-mono Suc-pred eq-iff le0)
    obtain  $C$  where  $\chi C: \chi = C + \{\#\text{Neg } v\# \}$  and  $\text{neg}C: \text{Neg } v \notin C$  and  $\text{pos}C: \text{Pos } v \notin C$ 
    proof -
      assume  $a1: \bigwedge C. [\chi = C + \{\#\text{Neg } v\# \}; \text{Neg } v \notin C; \text{Pos } v \notin C] \implies \text{thesis}$ 
      have  $f2: \bigwedge n. (0::\text{nat}) + n = n$ 
        by simp
      obtain  $mm :: 'v \text{ literal multiset} \Rightarrow 'v \text{ literal} \Rightarrow 'v \text{ literal multiset}$  where
         $f3: \{\#\text{Neg } v\# \} + mm \chi (\text{Neg } v) = \chi$ 
        by (metis (no-types) <count  $\chi (\text{Neg } v) = 1$ > add commute multi-member-split zero-less-one)
      then have  $\text{Pos } v \notin mm \chi (\text{Neg } v)$ 
        using  $f2$  by (metis (no-types)  $\text{Pos}v$  <count  $\chi (\text{Neg } v) = 1$ > add.right-neutral
          add-left-cancel count-single count-union less-nat-zero-code)
      then show ?thesis
    qed
  }
}

```

```

    using f3 a1 by (metis (no-types) ‹count  $\chi$  (Neg v) = 1› add.commute
      add.right-neutral add-left-cancel count-single count-union less-nat-zero-code)
  qed
obtain C' where
   $\chi C'$ :  $\chi' = C' + \{\#Pos\ v\#\}$  and
   $posC'$ :  $Pos\ v \notin\# C'$  and
   $negC'$ :  $Neg\ v \notin\# C'$ 
  by (metis (no-types, hide-lams) Negv ‹count  $\chi'$  (Pos v) = 1› add-diff-cancel-right'
    cancel-comm-monoid-add-class.diff-cancel count-diff count-single less-nat-zero-code
    mset-leD mset-le-add-left multi-member-split zero-less-one)

have totC: total-over-m I {C}
  using tot $\chi$  tot-over-m-remove[of I Pos v C] negC posC unfolding  $\chi C$ 
  by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
have totC': total-over-m I {C'}
  using tot $\chi'$  total-over-m-sum tot-over-m-remove[of I Neg v C'] negC' posC'
  unfolding  $\chi C'$  by (metis total-over-m-sum uminus-Neg)
have  $\neg I \models C + C'$ 
  using  $\chi\ \chi'\ \chi C\ \chi C'$  by auto
then have part-I- $\psi'''$ : partial-interps Leaf I (fst  $\psi \cup \{C + C'\}$ )
  using totC totC' ‹ $\neg I \models C + C'$ › by (metis Un-insert-right insertI1
    partial-interps.simps(1) total-over-m-sum)
{
  assume ( $\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C$ )  $\notin$  snd  $\psi$ 
  then have inf'': inference  $\psi$  (fst  $\psi \cup \{C + C'\}$ , snd  $\psi \cup \{(\chi', \chi)\}$ )
    by (metis  $\chi'\psi\ \chi C\ \chi C'\ \chi\psi$  add.commute inference-step prod.collapse resolution)
  obtain N' where full: full simplify (fst  $\psi \cup \{C + C'\}$ ) N'
    by (metis finite-simplified-full-simp fst-conv inf'' inference-preserves-finite
      local.finite)
  have resolution  $\psi$  (N', snd  $\psi \cup \{(\chi', \chi)\}$ )
    using resolution.intros(2)[OF - simp full, of snd  $\psi$  snd  $\psi \cup \{(\chi', \chi)\}$ ] inf''
    by (metis surjective-pairing)
  moreover have partial-interps Leaf I N'
    using full-simplify-preserve-partial-tree[OF full part-I- $\psi'''$ ] .
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case
    by (metis (no-types) prod.sel(1) rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
}
moreover {
  assume a: ( $\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C$ )  $\in$  snd  $\psi$ 
  then have ( $\exists \chi \in$  fst  $\psi$ . ( $\forall I$ . total-over-m I  $\{C + C'\} \longrightarrow$  total-over-m I  $\{\chi\}$ )
     $\wedge$  ( $\forall I$ . total-over-m I  $\{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C$ ))  $\vee$  tautology ( $C' + C$ )
  proof -
    obtain p where p: Pos p  $\in\#$  ( $\{\#Pos\ v\#\} + C') \wedge$  Neg p  $\in\#$  ( $\{\#Neg\ v\#\} + C$ )
       $\wedge$  ( $\exists \chi \in$  fst  $\psi$ . ( $\forall I$ . total-over-m I  $\{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})\}$ 
       $\longrightarrow$  total-over-m I  $\{\chi\}) \wedge$  ( $\forall I$ . total-over-m I  $\{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$ )  $\vee$  tautology ( $(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$ )
      using a by (blast intro: allE[OF a-u-i[unfolded subsumes-def Ball-def],
        of ( $\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C$ )])
    { assume p  $\neq v$ 
      then have Pos p  $\in\# C' \wedge$  Neg p  $\in\# C$  using p by force
      then have ?thesis by (metis add-gr-0 count-union tautology-Pos-Neg)
    }
  }
moreover {

```



```

    assume  $p = v$ 
    then have ?thesis using  $p$  by (metis add.commute add-diff-cancel-left')
  }
  ultimately show ?thesis by auto
qed
moreover {
  assume  $\exists \chi \in \text{fst } \psi. (\forall I. \text{total-over-}m \ I \ \{C+C'\} \longrightarrow \text{total-over-}m \ I \ \{\chi\})$ 
     $\wedge (\forall I. \text{total-over-}m \ I \ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
  then obtain  $\vartheta$  where
     $\vartheta: \vartheta \in \text{fst } \psi$  and
     $\text{tot-}\vartheta\text{-}CC': \forall I. \text{total-over-}m \ I \ \{C+C'\} \longrightarrow \text{total-over-}m \ I \ \{\vartheta\}$  and
     $\vartheta\text{-inv}: \forall I. \text{total-over-}m \ I \ \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$  by blast
  have partial-interps Leaf  $I$  (fst  $\psi$ )
    using tot- $\vartheta$ - $CC'$   $\vartheta$   $\vartheta\text{-inv}$  tot $C$  tot $C'$   $\hookrightarrow I \models C + C'$  total-over- $m$ -sum by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size  $xs$  unfolding  $xs$  by auto
  ultimately have ?case by blast
}
moreover {
  assume taut $CC'$ : tautology ( $C' + C$ )
  have total-over- $m \ I \ \{C'+C\}$  using tot $C$  tot $C'$  total-over- $m$ -sum by auto
  then have  $\neg$ tautology ( $C' + C$ )
    using  $\hookrightarrow I \models C + C'$  unfolding add.commute[of  $C \ C'$ ] total-over- $m$ -def
    unfolding tautology-def by auto
  then have False using taut $CC'$  unfolding tautology-def by auto
}
ultimately have ?case by auto
}
ultimately have ?case by auto
}
ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
  assume size-ag: sem-tree-size ag > 0
  have sem-tree-size ag < sem-tree-size  $xs$  unfolding  $xs$  by auto
  moreover have partial-interps ag ( $I \cup \{\text{Pos } v\}$ ) (fst  $\psi$ )
  and partad: partial-interps ad ( $I \cup \{\text{Neg } v\}$ ) (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding  $xs$  by metis+
  moreover
    have sem-tree-size ag < sem-tree-size  $xs \implies \text{finite (fst } \psi) \implies \text{already-used-inv } \psi$ 
       $\implies \text{partial-interps ag (} I \cup \{\text{Pos } v\} \text{) (fst } \psi) \implies \text{simplified (fst } \psi)$ 
       $\implies \exists \text{tree}' \ \psi'. \text{resolution}^{**} \ \psi \ \psi' \wedge \text{partial-interps tree' (} I \cup \{\text{Pos } v\} \text{) (fst } \psi')$ 
       $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size ag} \vee \text{sem-tree-size ag} = 0)$ 
    using IH[of ag  $I \cup \{\text{Pos } v\}$ ] by auto
  ultimately obtain  $\psi' :: 'v \text{ state}$  and  $\text{tree}' :: 'v \text{ sem-tree}$  where
    inf: resolution $^{**} \ \psi \ \psi'$ 
    and part: partial-interps tree' ( $I \cup \{\text{Pos } v\}$ ) (fst  $\psi'$ )
    and size: sem-tree-size tree' < sem-tree-size ag  $\vee$  sem-tree-size ag = 0
    using finite part rtranclp.rtrancl-refl a-u-i simp by blast

  have partial-interps ad ( $I \cup \{\text{Neg } v\}$ ) (fst  $\psi'$ )
    using rtranclp-resolution-preserve-partial-tree inf partad by fast
  then have partial-interps (Node  $v$  tree' ad)  $I$  (fst  $\psi'$ ) using part by auto
  then have ?case using inf size size-ag part unfolding  $xs$  by fastforce
}
moreover {

```

```

assume size-ad: sem-tree-size ad > 0
have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
moreover
  have
    partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
    partial-interps ad (I ∪ {Neg v}) (fst ψ)
    using part partial-interps.simps(2) unfolding xs by metis+
moreover have sem-tree-size ad < sem-tree-size xs  $\longrightarrow$  finite (fst ψ)  $\longrightarrow$  already-used-inv ψ
 $\longrightarrow$  ( partial-interps ad (I ∪ {Neg v}) (fst ψ)  $\longrightarrow$  simplified (fst ψ)
 $\longrightarrow$  ( $\exists$  tree' ψ'. resolution** ψ ψ' ∧ partial-interps tree' (I ∪ {Neg v}) (fst ψ')
 $\wedge$  (sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0)))
    using IH by blast
ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
  inf: resolution** ψ ψ'
  and part: partial-interps tree' (I ∪ {Neg v}) (fst ψ')
  and size: sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0
  using finite part rtranclp.rtrancl-refl a-u-i simp by blast

have partial-interps ag (I ∪ {Pos v}) (fst ψ')
  using rtranclp-resolution-preserve-partial-tree inf partag by fast
then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
then have ?case using inf size size-ad unfolding xs by fastforce
}
ultimately have ?case by auto
}
ultimately show ?case by auto
qed

```

**lemma** *resolution-completeness-inv*:

```

fixes ψ :: 'v :: linorder state
assumes
  unsat:  $\neg$ satisfiable (fst ψ) and
  finite: finite (fst ψ) and
  a-u-v: already-used-inv ψ
shows  $\exists \psi'. (resolution^{**} \psi \psi' \wedge \{\#\} \in fst \psi')$ 
proof –
obtain tree where partial-interps tree {} (fst ψ)
  using partial-interps-build-sem-tree-atms assms by metis
then show ?thesis
  using unsat finite a-u-v
proof (induct tree arbitrary: ψ rule: sem-tree-size)
  case (bigger tree ψ) note H = this
  {
    fix χ
    assume tree: tree = Leaf
    obtain χ where χ:  $\neg \{\} \models \chi$  and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
    using H unfolding tree by auto
    moreover have {#} = χ
    using H atms-empty-iff-empty totχ
    unfolding true-cls-def total-over-m-def total-over-set-def by fastforce
    moreover have resolution** ψ ψ by auto
    ultimately have ?case by metis
  }
moreover {
  fix v tree1 tree2

```

```

assume tree: tree = Node v tree1 tree2
obtain  $\psi_0$  where  $\psi_0$ : resolution**  $\psi$   $\psi_0$  and simp: simplified (fst  $\psi_0$ )
proof –
  { assume simplified (fst  $\psi$ )
    moreover have resolution**  $\psi$   $\psi$  by auto
    ultimately have thesis using that by blast
  }
moreover {
  assume  $\neg$ simplified (fst  $\psi$ )
  then have  $\exists \psi'$ . full1 simplify (fst  $\psi$ )  $\psi'$ 
    by (metis Nitpick.rtranclp-unfold bigger.prems(3) full1-def
      rtranclp-simplify-terminates)
  then obtain N where full1 simplify (fst  $\psi$ ) N by metis
  then have resolution  $\psi$  (N, snd  $\psi$ )
    using resolution.intros(1)[of fst  $\psi$  N snd  $\psi$ ] by auto
  moreover have simplified N
    using  $\langle$ full1 simplify (fst  $\psi$ ) N $\rangle$  unfolding full1-def by blast
  ultimately have ?thesis using that by force
  }
ultimately show ?thesis by auto
qed

```

```

have p: partial-interps tree {} (fst  $\psi_0$ )
and uns: unsatisfiable (fst  $\psi_0$ )
and f: finite (fst  $\psi_0$ )
and a-u-v: already-used-inv  $\psi_0$ 
  using  $\psi_0$  bigger.prems(1) rtranclp-resolution-preserve-partial-tree apply blast
  using  $\psi_0$  bigger.prems(2) rtranclp-resolution-preserves-unsat apply blast
  using  $\psi_0$  bigger.prems(3) rtranclp-resolution-finite apply blast
  using rtranclp-resolution-already-used-inv[OF  $\psi_0$  bigger.prems(4)] by blast
obtain tree'  $\psi'$  where
  inf: resolution**  $\psi_0$   $\psi'$  and
  part': partial-interps tree' {} (fst  $\psi'$ ) and
  decrease: sem-tree-size tree' < sem-tree-size tree  $\vee$  sem-tree-size tree = 0
  using can-decrease-tree-size-resolution[OF f a-u-v p simp] unfolding tautology-def
  by meson
have s: sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto
have fin: finite (fst  $\psi'$ )
  using f inf rtranclp-resolution-finite by blast
have unsat: unsatisfiable (fst  $\psi'$ )
  using rtranclp-resolution-preserves-unsat inf uns by metis
have a-u-i': already-used-inv  $\psi'$ 
  using a-u-v inf rtranclp-resolution-already-used-inv[of  $\psi_0$   $\psi'$ ] by auto
have ?case
  using inf rtranclp-trans[of resolution] H(1)[OF s part' unsat fin a-u-i']  $\psi_0$  by blast
}
ultimately show ?case by (case-tac tree, auto)
qed
qed

```

**lemma** *resolution-preserves-already-used-inv*:  
**assumes** *resolution* *S* *S'*  
**and** *already-used-inv* *S*  
**shows** *already-used-inv* *S'*

```

using assms
apply (induct rule: resolution.induct)
  apply (rule full1-simplify-already-used-inv; simp)
  apply (rule full-simplify-already-used-inv, simp)
  apply (rule inference-preserves-already-used-inv, simp)
  apply blast
done

lemma rtrancpl-resolution-preserves-already-used-inv:
  assumes resolution** S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
  apply (induct rule: rtrancpl-induct)
  apply simp
  using resolution-preserves-already-used-inv by fast

lemma resolution-completeness:
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes unsat:  $\neg \text{satisfiable (fst } \psi)$ 
  and finite:  $\text{finite (fst } \psi)$ 
  and snd  $\psi = \{\}$ 
  shows  $\exists \psi'. (\text{resolution** } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$ 
proof -
  have already-used-inv  $\psi$  unfolding assms by auto
  then show ?thesis using assms resolution-completeness-inv by blast
qed

lemma rtrancpl-preserves-sat:
  assumes simplify** S S'
  and satisfiable S
  shows satisfiable S'
  using assms apply induction
  apply simp
  by (meson satisfiable-carac satisfiable-def simplify-preserves-un-sat-eq)

lemma resolution-preserves-sat:
  assumes resolution S S'
  and satisfiable (fst S)
  shows satisfiable (fst S')
  using assms apply (induction rule: resolution.induct)
  using rtrancpl-preserves-sat trancpl-into-rtrancpl unfolding full1-def apply fastforce
  by (metis fst-conv full-def inference-preserves-un-sat rtrancpl-preserves-sat
    satisfiable-carac' satisfiable-def)

lemma rtrancpl-resolution-preserves-sat:
  assumes resolution** S S'
  and satisfiable (fst S)
  shows satisfiable (fst S')
  using assms apply (induction rule: rtrancpl-induct)
  apply simp
  using resolution-preserves-sat by blast

lemma resolution-soundness:
  fixes  $\psi :: 'v :: \text{linorder state}$ 

```

```

assumes resolution**  $\psi \ \psi'$  and  $\{\#\} \in \text{fst } \psi'$ 
shows unsatisfiable (fst  $\psi$ )
using assms by (meson rtrancp-resolution-preserves-sat satisfiable-def true-cls-empty
  true-cls-def)

```

```

lemma resolution-soundness-and-completeness:
fixes  $\psi :: 'v :: \text{linorder state}$ 
assumes finite: finite (fst  $\psi$ )
and snd: snd  $\psi = \{\}$ 
shows  $(\exists \psi'. (\text{resolution** } \psi \ \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable } (\text{fst } \psi)$ 
using assms resolution-completeness resolution-soundness by metis

```

```

lemma simplified-falsity:
assumes simp: simplified  $\psi$ 
and  $\{\#\} \in \psi$ 
shows  $\psi = \{\{\#\}\}$ 
proof (rule ccontr)
assume H:  $\neg ?thesis$ 
then obtain  $\chi$  where  $\chi \in \psi$  and  $\chi \neq \{\#\}$  using assms(2) by blast
then have  $\{\#\} \subset \# \chi$  by (simp add: mset-less-empty-nonempty)
then have simplify  $\psi$  ( $\psi - \{\chi\}$ )
using simplify.subsumption[OF assms(2)  $\langle \{\#\} \subset \# \chi \rangle \langle \chi \in \psi \rangle$ ] by blast
then show False using simp by blast
qed

```

```

lemma simplify-falsity-in-preserved:
assumes simplify  $\chi s \ \chi s'$ 
and  $\{\#\} \in \chi s$ 
shows  $\{\#\} \in \chi s'$ 
using assms
by induction auto

```

```

lemma rtrancp-simplify-falsity-in-preserved:
assumes simplify**  $\chi s \ \chi s'$ 
and  $\{\#\} \in \chi s$ 
shows  $\{\#\} \in \chi s'$ 
using assms
by induction (auto intro: simplify-falsity-in-preserved)

```

```

lemma resolution-falsity-get-falsity-alone:
assumes finite (fst  $\psi$ )
shows  $(\exists \psi'. (\text{resolution** } \psi \ \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow (\exists a-u-v. \text{resolution** } \psi (\{\{\#\}\}, a-u-v))$ 
  (is  $?A \longleftrightarrow ?B$ )
proof
assume  $?B$ 
then show  $?A$  by auto
next
assume  $?A$ 
then obtain  $\chi s \ a-u-v$  where  $\chi s$ : resolution**  $\psi (\chi s, a-u-v)$  and  $F$ :  $\{\#\} \in \chi s$  by auto
{ assume simplified  $\chi s$ 
  then have  $?B$  using simplified-falsity[OF - F]  $\chi s$  by blast
}
moreover {
  assume  $\neg \text{simplified } \chi s$ 

```

```

then obtain  $\chi s'$  where full1 simplify  $\chi s \chi s'$ 
  by (metis  $\chi s$  assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite)
then have  $\{\#\} \in \chi s'$ 
  unfolding full1-def by (meson  $F$  rtranclp-simplify-falsity-in-preserved
    trancplp-into-rtranclp)
then have ? $B$ 
  by (metis  $\chi s$  (full1 simplify  $\chi s \chi s'$ ) fst-conv full1-simp resolution-always-simplified
    rtranclp.rtrancl-into-rtrancl simplified-falsity)
}
ultimately show ? $B$  by blast
qed

lemma resolution-soundness-and-completeness':
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes
    finite: finite (fst  $\psi$ ) and
    snd: snd  $\psi = \{\}$ 
  shows  $(\exists a-u-v. (\text{resolution}^{**} \psi (\{\#\}, a-u-v))) \longleftrightarrow \text{unsatisfiable} (\text{fst } \psi)$ 
  using assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone
  by metis

end

theory Partial-Annotated-Clausal-Logic
imports Partial-Clausal-Logic

begin

```

## 13 Partial Clausal Logic

We here define marked literals (that will be used in both DPLL and CDCL) and the entailment corresponding to it.

### 13.1 Marked Literals

#### 13.1.1 Definition

```

datatype ( $'v, 'l, 'mark$ ) marked-lit =
  is-marked: Marked (lit-of:  $'v$  literal) (level-of:  $'l$ ) |
  is-proped: Propagated (lit-of:  $'v$  literal) (mark-of:  $'mark$ )

lemma marked-lit-list-induct[case-names nil marked proped]:
  assumes  $P []$  and
   $\bigwedge L l xs. P xs \implies P (\text{Marked } L l \# xs)$  and
   $\bigwedge L m xs. P xs \implies P (\text{Propagated } L m \# xs)$ 
  shows  $P xs$ 
  using assms apply (induction xs, simp)
  by (case-tac a) auto

lemma is-marked-ex-Marked:
   $\text{is-marked } L \implies \exists K l. L = \text{Marked } K l$ 
  by (cases L) auto

type-synonym ( $'v, 'l, 'm$ ) marked-lits = ( $'v, 'l, 'm$ ) marked-lit list

```

**definition** *lits-of* :: ('a, 'b, 'c) marked-lit list  $\Rightarrow$  'a literal set **where**  
*lits-of* Ls = *lit-of* ' (set Ls)

**lemma** *lits-of-empty[simp]*:  
*lits-of* [] = {} **unfolding** *lits-of-def* **by** *auto*

**lemma** *lits-of-cons[simp]*:  
*lits-of* (L # Ls) = *insert* (*lit-of* L) (*lits-of* Ls)  
**unfolding** *lits-of-def* **by** *auto*

**lemma** *lits-of-append[simp]*:  
*lits-of* (l @ l') = *lits-of* l  $\cup$  *lits-of* l'  
**unfolding** *lits-of-def* **by** *auto*

**lemma** *finite-lits-of-def[simp]*: *finite* (*lits-of* L)  
**unfolding** *lits-of-def* **by** *auto*

**lemma** *lits-of-rev[simp]*: *lits-of* (rev M) = *lits-of* M  
**unfolding** *lits-of-def* **by** *auto*

**lemma** *set-map-lit-of-lits-of[simp]*:  
*set* (map *lit-of* T) = *lits-of* T  
**unfolding** *lits-of-def* **by** *auto*

**lemma** *atms-of-m-lambda-lit-of-is-atm-of-lit-of[simp]*:  
*atms-of-m* (( $\lambda a. \{\# \text{lit-of } a \# \}$ ) ' set M') = *atm-of* ' *lits-of* M'  
**unfolding** *atms-of-m-def* *lits-of-def* **by** *auto*

**lemma** *lits-of-empty-is-empty[iff]*:  
*lits-of* M = {}  $\longleftrightarrow$  M = []  
**by** (*induct* M) *auto*

### 13.1.2 Entailment

**definition** *true-annot* :: ('a, 'l, 'm) marked-lits  $\Rightarrow$  'a clause  $\Rightarrow$  bool (**infix**  $\models_a$  49) **where**  
 $I \models_a C \longleftrightarrow (\text{lits-of } I) \models C$

**definition** *true-annots* :: ('a, 'l, 'm) marked-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]*:  
 $\neg [] \models_a \psi$   
**unfolding** *true-annot-def* *true-cls-def* **by** *simp*

**lemma** *true-annot-empty[simp]*:  
 $\neg I \models_a \{\#\}$   
**unfolding** *true-annot-def* *true-cls-def* **by** *simp*

**lemma** *empty-true-annots-def[iff]*:  
 $[] \models_{as} \psi \longleftrightarrow \psi = \{\}$   
**unfolding** *true-annots-def* **by** *auto*

**lemma** *true-annots-empty[simp]*:  
 $I \models_{as} \{\}$   
**unfolding** *true-annots-def* **by** *auto*

**lemma** *true-annots-single-true-annot*[*iff*]:

$I \models_{as} \{C\} \longleftrightarrow I \models_a C$

**unfolding** *true-annots-def* **by** *auto*

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

$M \models_a A \implies L \# M \models_a A$

**unfolding** *true-annot-def* **by** *auto*

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

$M \models_{as} A \implies L \# M \models_{as} A$

**unfolding** *true-annots-def* **by** *auto*

**lemma** *true-annots-union*[*iff*]:

$M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$

**unfolding** *true-annots-def* **by** *auto*

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

$M \models_{as} \text{insert } a \ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$

**unfolding** *true-annots-def* **by** *auto*

Link between  $\models_{as}$  and  $\models_s$ :

**lemma** *true-annots-true-cls*:

$I \models_{as} CC \longleftrightarrow (\text{lits-of } I) \models_s CC$

**unfolding** *true-annots-def* *Ball-def* *true-annot-def* *true-clss-def* **by** *auto*

**lemma** *in-lit-of-true-annot*:

$a \in \text{lits-of } M \longleftrightarrow M \models_a \{\#a\# \}$

**unfolding** *true-annot-def* *lits-of-def* **by** *auto*

**lemma** *true-annot-lit-of-notin-skip*:

$L \# M \models_a A \implies \text{lit-of } L \notin \# A \implies M \models_a A$

**unfolding** *true-annot-def* *true-cls-def* **by** *auto*

**lemma** *true-clss-singleton-lit-of-implies-incl*:

$I \models_s (\lambda a. \{\#\text{lit-of } a\#\}) \text{ 'set MLs } \implies \text{lits-of MLs} \subseteq I$

**unfolding** *true-clss-def* *lits-of-def* **by** *auto*

**lemma** *true-annot-true-clss-cls*:

$MLs \models_a \psi \implies \text{set } (\text{map } (\lambda a. \{\#\text{lit-of } a\#\}) \ MLs) \models_p \psi$

**unfolding** *true-annot-def* *true-clss-cls-def* *true-cls-def*

**by** (*auto* *dest: true-clss-singleton-lit-of-implies-incl*)

**lemma** *true-annots-true-clss-cls*:

$MLs \models_{as} \psi \implies \text{set } (\text{map } (\lambda a. \{\#\text{lit-of } a\#\}) \ MLs) \models_{ps} \psi$

**by** (*auto*

*dest: true-clss-singleton-lit-of-implies-incl*

*simp add: true-clss-def true-annots-def true-annot-def lits-of-def true-cls-def*

*true-clss-clss-def*)

**lemma** *true-annots-marked-true-cls*[*iff*]:

$\text{map } (\lambda M. \text{Marked } M \ a) \ M \models_{as} N \longleftrightarrow \text{set } M \models_s N$

**proof** –

**have** \*: *lits-of* (*map* ( $\lambda M. \text{Marked } M \ a$ ) *M*) = *set M* **unfolding** *lits-of-def* **by** *force*

**show** ?*thesis* **by** (*simp add: true-annots-true-cls* \*)



qed

**lemma** *true-annot-singleton[iff]*:  $M \models_a \{\#L\# \} \longleftrightarrow L \in \text{lits-of } M$   
**unfolding** *true-annot-def lits-of-def* **by** *auto*

**lemma** *true-annots-true-clss-clss*:  
 $A \models_{as} \Psi \implies (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } A \models_{ps} \Psi$   
**unfolding** *true-clss-clss-def true-annots-def true-clss-def*  
**by** (*auto*  
*dest!*: *true-clss-singleton-lit-of-implies-incl*  
*simp add*: *lits-of-def true-annot-def true-clss-def*)

**lemma** *true-annot-commute*:  
 $M @ M' \models_a D \longleftrightarrow M' @ M \models_a D$   
**unfolding** *true-annot-def* **by** (*simp add*: *Un-commute*)

**lemma** *true-annots-commute*:  
 $M @ M' \models_{as} D \longleftrightarrow M' @ M \models_{as} D$   
**unfolding** *true-annots-def* **by** (*auto simp add*: *true-annot-commute*)

**lemma** *true-annot-mono[dest]*:  
 $\text{set } I \subseteq \text{set } I' \implies I \models_a N \implies I' \models_a N$   
**using** *true-clss-mono-set-mset-l* **unfolding** *true-annot-def lits-of-def*  
**by** (*metis (no-types) Un-commute Un-upper1 image-Un sup.orderE*)

**lemma** *true-annots-mono*:  
 $\text{set } I \subseteq \text{set } I' \implies I \models_{as} N \implies I' \models_{as} N$   
**unfolding** *true-annots-def* **by** *auto*

### 13.1.3 Defined and undefined literals

**definition** *defined-lit* :: ('a, 'l, 'm) *marked-lit list*  $\Rightarrow$  'a *literal*  $\Rightarrow$  bool  
**where**  
 $\text{defined-lit } I L \longleftrightarrow (\exists l. \text{Marked } L l \in \text{set } I) \vee (\exists P. \text{Propagated } L P \in \text{set } I)$   
 $\vee (\exists l. \text{Marked } (-L) l \in \text{set } I) \vee (\exists P. \text{Propagated } (-L) P \in \text{set } I)$

**abbreviation** *undefined-lit* :: ('a, 'l, 'm) *marked-lit list*  $\Rightarrow$  'a *literal*  $\Rightarrow$  bool  
**where** *undefined-lit*  $I L \equiv \neg \text{defined-lit } I L$

**lemma** *defined-lit-rev[simp]*:  
 $\text{defined-lit } (\text{rev } M) L \longleftrightarrow \text{defined-lit } M L$   
**unfolding** *defined-lit-def* **by** *auto*

**lemma** *atm-imp-marked-or-proped*:  
**assumes**  $x \in \text{set } I$   
**shows**  
 $(\exists l. \text{Marked } (- \text{lit-of } x) l \in \text{set } I)$   
 $\vee (\exists l. \text{Marked } (\text{lit-of } x) l \in \text{set } I)$   
 $\vee (\exists l. \text{Propagated } (- \text{lit-of } x) l \in \text{set } I)$   
 $\vee (\exists l. \text{Propagated } (\text{lit-of } x) l \in \text{set } I)$   
**using** *assms marked-lit.exhaust-sel* **by** *metis*

**lemma** *literal-is-lit-of-marked*:  
**assumes**  $L = \text{lit-of } x$   
**shows**  $(\exists l. x = \text{Marked } L l) \vee (\exists l'. x = \text{Propagated } L l')$   
**using** *assms* **by** (*case-tac x*) *auto*

**lemma** *true-annot-iff-marked-or-true-lit*:  
 $\text{defined-lit } I \ L \longleftrightarrow ((\text{lits-of } I) \models_l L \vee (\text{lits-of } I) \models_l \neg L)$   
**unfolding** *defined-lit-def* **by** (*auto simp add: lits-of-def rev-image-eqI dest!: literal-is-lit-of-marked*)

**lemma** *consistent-interp* ( $\text{lits-of } I \implies I \models_{as} N \implies \text{satisfiable } N$ )  
**by** (*simp add: true-annots-true-cls*)

**lemma** *defined-lit-map*:  
 $\text{defined-lit } Ls \ L \longleftrightarrow \text{atm-of } L \in (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } Ls$   
**unfolding** *defined-lit-def* **apply** (*rule iffI*)  
**using** *image-iff* **apply** *fastforce*  
**by** (*fastforce simp add: atm-of-eq-atm-of dest: atm-imp-marked-or-proped*)

**lemma** *defined-lit-uminus*[*iff*]:  
 $\text{defined-lit } I \ (-L) \longleftrightarrow \text{defined-lit } I \ L$   
**unfolding** *defined-lit-def* **by** *auto*

**lemma** *Marked-Propagated-in-iff-in-lits-of*:  
 $\text{defined-lit } I \ L \longleftrightarrow (L \in \text{lits-of } I \vee \neg L \in \text{lits-of } I)$   
**unfolding** *lits-of-def* *defined-lit-def*  
**by** (*auto simp add: rev-image-eqI*) (*case-tac x, auto*)+

**lemma** *consistent-add-undefined-lit-consistent*[*simp*]:  
**assumes**  
 $\text{consistent-interp } (\text{lits-of } Ls) \text{ and }$   
 $\text{undefined-lit } Ls \ L$   
**shows**  $\text{consistent-interp } (\text{insert } L \ (\text{lits-of } Ls))$   
**using** *assms* **unfolding** *consistent-interp-def* **by** (*auto simp: Marked-Propagated-in-iff-in-lits-of*)

**lemma** *decided-empty*[*simp*]:  
 $\neg \text{defined-lit } [] \ L$   
**unfolding** *defined-lit-def* **by** *simp*

## 13.2 Backtracking

**fun** *backtrack-split* :: (*'v, 'l, 'm*) *marked-lits*  
 $\Rightarrow (\text{'v, 'l, 'm}) \text{ marked-lits} \times (\text{'v, 'l, 'm}) \text{ marked-lits}$  **where**  
 $\text{backtrack-split } [] = ([], [])$  |  
 $\text{backtrack-split } (\text{Propagated } L \ P \ \# \ \text{mlits}) = \text{apfst } ((\text{op } \#) \ (\text{Propagated } L \ P)) \ (\text{backtrack-split } \text{mlits})$  |  
 $\text{backtrack-split } (\text{Marked } L \ l \ \# \ \text{mlits}) = ([], \text{Marked } L \ l \ \# \ \text{mlits})$

**lemma** *backtrack-split-fst-not-marked*:  $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \implies \neg \text{is-marked } a$   
**by** (*induct l rule: marked-lit-list-induct*) *auto*

**lemma** *backtrack-split-snd-hd-marked*:  
 $\text{snd } (\text{backtrack-split } l) \neq [] \implies \text{is-marked } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$   
**by** (*induct l rule: marked-lit-list-induct*) *auto*

**lemma** *backtrack-split-list-eq*[*simp*]:  
 $\text{fst } (\text{backtrack-split } l) @ (\text{snd } (\text{backtrack-split } l)) = l$   
**by** (*induct l rule: marked-lit-list-induct*) *auto*

**lemma** *backtrack-split-snd-empty-not-marked*:  
 $\text{backtrack-split } M = (M'', []) \implies \forall l \in \text{set } M. \neg \text{is-marked } l$

by (metis append-Nil2 backtrack-split-fst-not-marked backtrack-split-list-eq snd-conv)

**lemma** *backtrack-split-some-is-marked-then-snd-has-hd*:

$\exists l \in \text{set } M. \text{ is-marked } l \implies \exists M' L' M''. \text{ backtrack-split } M = (M', L' \# M')$

by (metis backtrack-snd-empty-not-marked list.exhaust prod.collapse)

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*:

$\text{backtrack-split } M = (\text{takeWhile } (\text{Not } o \text{ is-marked}) M, \text{dropWhile } (\text{Not } o \text{ is-marked}) M)$

**proof** (induct M)

case Nil show ?case by simp

next

case (Cons L M) thus ?case by (cases L) auto

qed

### 13.3 Decomposition with respect to the marked literals

The pattern *get-all-marked-decomposition*  $\square = [(\square, \square)]$  is necessary otherwise, we can call the *hd* function in the other pattern.

**fun** *get-all-marked-decomposition* :: ('a, 'l, 'm) marked-lits

$\Rightarrow ((\text{'a, 'l, 'm marked-lits} \times \text{'a, 'l, 'm marked-lits}) \text{ list where}$

*get-all-marked-decomposition* (Marked L l # Ls) =

(Marked L l # Ls,  $\square$ ) # *get-all-marked-decomposition* Ls |

*get-all-marked-decomposition* (Propagated L P # Ls) =

(apsnd ((op #) (Propagated L P)) (hd (*get-all-marked-decomposition* Ls)))

# tl (*get-all-marked-decomposition* Ls) |

*get-all-marked-decomposition*  $\square = [(\square, \square)]$

**value** *get-all-marked-decomposition* [Propagated A5 B5, Marked C4 D4, Propagated A3 B3,  
Propagated A2 B2, Marked C1 D1, Propagated A0 B0]

**lemma** *get-all-marked-decomposition-never-empty[iff]*:

*get-all-marked-decomposition* M =  $\square \iff \text{False}$

by (induct M, simp) (case-tac a, auto)

**lemma** *get-all-marked-decomposition-never-empty-sym[iff]*:

$\square = \text{get-all-marked-decomposition } M \iff \text{False}$

using *get-all-marked-decomposition-never-empty[of M]* by presburger

**lemma** *get-all-marked-decomposition-decomp*:

$\text{hd } (\text{get-all-marked-decomposition } S) = (a, c) \implies S = c @ a$

**proof** (induct S arbitrary: a c)

case Nil

thus ?case by simp

next

case (Cons x A)

thus ?case by (cases x; cases hd (*get-all-marked-decomposition* A)) auto

qed

**lemma** *get-all-marked-decomposition-backtrack-split*:

$\text{backtrack-split } S = (M, M') \iff \text{hd } (\text{get-all-marked-decomposition } S) = (M', M)$

**proof** (induction S arbitrary: M M')

case Nil

```

thus ?case by auto
next
  case (Cons a S)
  thus ?case using backtrack-split-takeWhile-dropWhile by (cases a) force+
qed

lemma get-all-marked-decomposition-nil-backtrack-split-snd-nil:
  get-all-marked-decomposition S = [([], A)]  $\implies$  snd (backtrack-split S) = []
  by (simp add: get-all-marked-decomposition-backtrack-split sndI)

lemma get-all-marked-decomposition-length-1-fst-empty-or-length-1:
  assumes get-all-marked-decomposition M = (a, b) # []
  shows a = []  $\vee$  (length a = 1  $\wedge$  is-marked (hd a)  $\wedge$  hd a  $\in$  set M)
  using assms
proof (induct M arbitrary: a b)
  case Nil thus ?case by simp
next
  case (Cons m M)
  show ?case
  proof (cases m)
  case (Marked l mark)
  thus ?thesis using Cons by simp
  next
  case (Propagated l mark)
  thus ?thesis using Cons by (cases get-all-marked-decomposition M) force+
  qed
qed

lemma get-all-marked-decomposition-fst-empty-or-hd-in-M:
  assumes get-all-marked-decomposition M = (a, b) # l
  shows a = []  $\vee$  (is-marked (hd a)  $\wedge$  hd a  $\in$  set M)
  using assms apply (induct M arbitrary: a b rule: marked-lit-list-induct)
  apply auto[2]
  by (metis UnCI backtrack-split-snd-hd-marked get-all-marked-decomposition-backtrack-split
    get-all-marked-decomposition-decomp hd-in-set list.sel(1) set-append snd-conv)

lemma get-all-marked-decomposition-snd-not-marked:
  assumes (a, b)  $\in$  set (get-all-marked-decomposition M)
  and L  $\in$  set b
  shows  $\neg$ is-marked L
  using assms apply (induct M arbitrary: a b rule: marked-lit-list-induct, simp)
  by (case-tac get-all-marked-decomposition xs; fastforce)+

lemma tl-get-all-marked-decomposition-skip-some:
  assumes x  $\in$  set (tl (get-all-marked-decomposition M1))
  shows x  $\in$  set (tl (get-all-marked-decomposition (M0 @ M1)))
  using assms
  by (induct M0 rule: marked-lit-list-induct)
  (auto simp add: list.set-sel(2))

lemma hd-get-all-marked-decomposition-skip-some:
  assumes (x, y) = hd (get-all-marked-decomposition M1)
  shows (x, y)  $\in$  set (get-all-marked-decomposition (M0 @ Marked K i # M1))
  using assms
proof (induct M0)

```

```

case Nil
thus ?case by auto
next
case (Cons L M0)
hence xy: (x, y) ∈ set (get-all-marked-decomposition (M0 @ Marked K i # M1)) by blast
show ?case
proof (cases L)
case (Marked l m)
thus ?thesis using xy by auto
next
case (Propagated l m)
thus ?thesis
using xy Cons.prem by
by (cases get-all-marked-decomposition (M0 @ Marked K i # M1))
(auto dest!: get-all-marked-decomposition-decomp
arg-cong[of get-all-marked-decomposition - - hd])
qed
qed

lemma get-all-marked-decomposition-snd-union:
set M =  $\bigcup$  (set 'snd ' set (get-all-marked-decomposition M))  $\cup$  {L | L. is-marked L  $\wedge$  L ∈ set M}
(is ?M M = ?U M  $\cup$  ?Ls M)
proof (induct M arbitrary:)
case Nil
thus ?case by simp
next
case (Cons L M)
show ?case
proof (cases L)
case (Marked a l) note L = this
hence L ∈ ?Ls (L#M) by auto
moreover have ?U (L#M) = ?U M unfolding L by auto
moreover have ?M M = ?U M  $\cup$  ?Ls M using Cons.hyps by auto
ultimately show ?thesis by auto
next
case (Propagated a P)
thus ?thesis using Cons.hyps by (cases (get-all-marked-decomposition M)) auto
qed
qed

lemma in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend:
(a, b) ∈ set (get-all-marked-decomposition M')  $\implies$ 
 $\exists b'. (a, b' @ b) \in \text{set (get-all-marked-decomposition (M @ M'))}$ 
apply (induction M rule: marked-lit-list-induct)
apply (metis append-Nil)
apply auto[]
by (case-tac get-all-marked-decomposition (xs @ M')) auto

lemma get-all-marked-decomposition-remove-unmarked-length:
assumes  $\forall l \in \text{set } M'. \neg \text{is-marked } l$ 
shows length (get-all-marked-decomposition (M' @ M''))
= length (get-all-marked-decomposition M'')
using assms by (induct M' arbitrary: M'' rule: marked-lit-list-induct) auto

lemma get-all-marked-decomposition-not-is-marked-length:

```

**assumes**  $\forall l \in \text{set } M'. \neg \text{is-marked } l$   
**shows**  $1 + \text{length } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$   
 $= \text{length } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M))$   
**using** *assms get-all-marked-decomposition-remove-unmarked-length* **by** *fastforce*

**lemma** *get-all-marked-decomposition-last-choice*:

**assumes**  $tl (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)) \neq []$   
**and**  $\forall l \in \text{set } M'. \neg \text{is-marked } l$   
**and**  $hd (tl (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M))) = (M0', M0)$   
**shows**  $hd (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$   
**using** *assms* **by** (*induct M' rule: marked-lit-list-induct*) *auto*

**lemma** *get-all-marked-decomposition-except-last-choice-equal*:

**assumes**  $\forall l \in \text{set } M'. \neg \text{is-marked } l$   
**shows**  $tl (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$   
 $= tl (tl (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)))$   
**using** *assms* **by** (*induct M' rule: marked-lit-list-induct*) *auto*

**lemma** *get-all-marked-decomposition-hd-hd*:

**assumes**  $\text{get-all-marked-decomposition } Ls = (M, C) \# (M0, M0') \# l$   
**shows**  $tl M = M0' @ M0 \wedge \text{is-marked } (hd M)$   
**using** *assms*

**proof** (*induct Ls arbitrary: M C M0 M0' l*)

**case** *Nil*  
**thus** *?case* **by** *simp*

**next**

**case** (*Cons a Ls M C M0 M0' l*) **note**  $IH = \text{this}(1)$  **and**  $g = \text{this}(2)$   
**{** **fix**  $L \text{ level}$   
**assume**  $a: a = \text{Marked } L \text{ level}$   
**have**  $Ls = M0' @ M0$   
**using**  $g \text{ a by } (\text{force intro: get-all-marked-decomposition-decomp})$   
**hence**  $tl M = M0' @ M0 \wedge \text{is-marked } (hd M)$  **using**  $g \text{ a by auto}$   
**}**

**moreover** **{**  
**fix**  $L P$   
**assume**  $a: a = \text{Propagated } L P$   
**have**  $tl M = M0' @ M0 \wedge \text{is-marked } (hd M)$   
**using**  $IH \text{ Cons.premis unfolding a by } (\text{cases get-all-marked-decomposition } Ls) \text{ auto}$   
**}**

**ultimately show** *?case* **by** (*cases a*) *auto*

**qed**

**lemma** *get-all-marked-decomposition-exists-prepend[dest]*:

**assumes**  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$   
**shows**  $\exists c. M = c @ b @ a$   
**using** *assms* **apply** (*induct M rule: marked-lit-list-induct*)  
**apply** *simp*  
**by** (*case-tac get-all-marked-decomposition xs;*  
*auto dest!: arg-cong[of get-all-marked-decomposition - - hd]*  
*get-all-marked-decomposition-decomp*)**+**

**lemma** *get-all-marked-decomposition-incl*:

**assumes**  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$   
**shows**  $\text{set } b \subseteq \text{set } M$  **and**  $\text{set } a \subseteq \text{set } M$

**using** *assms* *get-all-marked-decomposition-exists-prepend* **by** *fastforce+*

**lemma** *get-all-marked-decomposition-exists-prepend'*:  
**assumes**  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$   
**obtains**  $c$  **where**  $M = c @ b @ a$   
**using** *assms* **apply** (*induct*  $M$  *rule*: *marked-lit-list-induct*)  
**apply** *auto*[1]  
**by** (*case-tac* *hd* (*get-all-marked-decomposition* *xs*),  
*auto* *dest*!: *get-all-marked-decomposition-decomp simp add: list.set-sel*(2))+

**lemma** *union-in-get-all-marked-decomposition-is-subset*:  
**assumes**  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$   
**shows**  $\text{set } a \cup \text{set } b \subseteq \text{set } M$   
**using** *assms* **by** *force*

**definition** *all-decomposition-implies* :: '*a* literal multiset set  
 $\Rightarrow ((\text{'a}, \text{'l}, \text{'m}) \text{ marked-lit list} \times (\text{'a}, \text{'l}, \text{'m}) \text{ marked-lit list}) \text{ list} \Rightarrow \text{bool}$  **where**  
*all-decomposition-implies*  $N \ S$   
 $\longleftrightarrow (\forall (Ls, \text{seen}) \in \text{set } S. (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } Ls \cup N \models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set seen})$

**lemma** *all-decomposition-implies-empty*[*iff*]:  
*all-decomposition-implies*  $N \ []$  **unfolding** *all-decomposition-implies-def* **by** *auto*

**lemma** *all-decomposition-implies-single*[*iff*]:  
*all-decomposition-implies*  $N \ [(Ls, \text{seen})]$   
 $\longleftrightarrow (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } Ls \cup N \models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set seen}$   
**unfolding** *all-decomposition-implies-def* **by** *auto*

**lemma** *all-decomposition-implies-append*[*iff*]:  
*all-decomposition-implies*  $N \ (S @ S')$   
 $\longleftrightarrow (\text{all-decomposition-implies } N \ S \wedge \text{all-decomposition-implies } N \ S')$   
**unfolding** *all-decomposition-implies-def* **by** *auto*

**lemma** *all-decomposition-implies-cons-pair*[*iff*]:  
*all-decomposition-implies*  $N \ ((Ls, \text{seen}) \# S')$   
 $\longleftrightarrow (\text{all-decomposition-implies } N \ [(Ls, \text{seen})] \wedge \text{all-decomposition-implies } N \ S')$   
**unfolding** *all-decomposition-implies-def* **by** *auto*

**lemma** *all-decomposition-implies-cons-single*[*iff*]:  
*all-decomposition-implies*  $N \ (l \# S') \longleftrightarrow$   
 $((\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } (\text{fst } l) \cup N \models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } (\text{snd } l) \wedge$   
*all-decomposition-implies*  $N \ S')$   
**unfolding** *all-decomposition-implies-def* **by** *auto*

**lemma** *all-decomposition-implies-trail-is-implied*:  
**assumes** *all-decomposition-implies*  $N \ (\text{get-all-marked-decomposition } M)$   
**shows**  $N \cup \{\{\# \text{lit-of } L \# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$   
 $\models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' } \bigcup (\text{set ' snd ' set } (\text{get-all-marked-decomposition } M))$   
**using** *assms*  
**proof** (*induct* *length* (*get-all-marked-decomposition*  $M$ ) *arbitrary*:  $M$ )  
**case** 0  
**thus** ?*case* **by** *auto*  
**next**  
**case** (*Suc*  $n$ ) **note**  $IH = \text{this}(1)$  **and**  $\text{length} = \text{this}(2)$

```

{
  assume length (get-all-marked-decomposition M) ≤ 1
  then obtain a b where g: get-all-marked-decomposition M = (a, b) # []
    by (case-tac get-all-marked-decomposition M) auto
  moreover {
    assume a = []
    hence ?case using Suc.premis g by auto
  }
  moreover {
    assume l: length a = 1 and m: is-marked (hd a) and hd: hd a ∈ set M
    hence (λa. {#lit-of a#}) (hd a) ∈ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M} by auto
    hence H: (λa. {#lit-of a#}) ' set a ∪ N ⊆ N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M}
      using l by (cases a) auto
    have f1: (λm. {#lit-of m#}) ' set a ∪ N ⊨ps (λm. {#lit-of m#}) ' set b
      using Suc.premis unfolding all-decomposition-implies-def g by simp
    have ?case
      unfolding g apply (rule true-clss-clss-subset) using f1 H by auto
  }
  ultimately have ?case using get-all-marked-decomposition-length-1-fst-empty-or-length-1 by blast
}
moreover {
  assume length (get-all-marked-decomposition M) > 1
  then obtain Ls0 seen0 M' where
    Ls0: get-all-marked-decomposition M = (Ls0, seen0) # get-all-marked-decomposition M' and
    length': length (get-all-marked-decomposition M') = n and
    M'-in-M: set M' ⊆ set M
    using length apply (induct M)
      apply simp
    by (case-tac a, case-tac hd (get-all-marked-decomposition M))
      (auto simp add: subset-insertI2)
  {
    assume n = 0
    hence get-all-marked-decomposition M' = [] using length' by auto
    hence ?case using Suc.premis unfolding all-decomposition-implies-def Ls0 by auto
  }
  moreover {
    assume n: n > 0
    then obtain Ls1 seen1 l where Ls1: get-all-marked-decomposition M' = (Ls1, seen1) # l
      using length' by (induct M', simp) (case-tac a, auto)

    have all-decomposition-implies N (get-all-marked-decomposition M')
      using Suc.premis unfolding Ls0 all-decomposition-implies-def by auto
    hence N: N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M'}
      ⊨ps (λa. {#lit-of a#}) ' ⋃ (set ' snd ' set (get-all-marked-decomposition M'))
      using IH length' by auto

    have l: N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M'}
      ⊆ N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M}
      using M'-in-M by auto
    hence ΨN: N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M}
      ⊨ps (λa. {#lit-of a#}) ' ⋃ (set ' snd ' set (get-all-marked-decomposition M'))
      using true-clss-clss-subset[OF l N] by auto
    have is-marked (hd Ls0) and LS: tl Ls0 = seen1 @ Ls1
      using get-all-marked-decomposition-hd-hd[of M] unfolding Ls0 Ls1 by auto
  }
}

```



```

have LSM: seen1 @ Ls1 = M' using get-all-marked-decomposition-decomp[of M'] Ls1 by auto
have M': set M' = Union (set 'snd ' set (get-all-marked-decomposition M'))
  ∪ {L | L. is-marked L ∧ L ∈ set M'}
  using get-all-marked-decomposition-snd-union by auto

{
  assume Ls0 ≠ []
  hence hd Ls0 ∈ set M using get-all-marked-decomposition-fst-empty-or-hd-in-M Ls0 by blast
  hence N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M} ⊨p (λa. {#lit-of a#}) (hd Ls0)
    using (is-marked (hd Ls0)) by (metis (mono-tags, lifting) UnCI mem-Collect-eq
      true-clss-clss-in)
} note hd-Ls0 = this

have l: (λa. {#lit-of a#}) ' (∪ (set 'snd ' set (get-all-marked-decomposition M'))
  ∪ {L | L. is-marked L ∧ L ∈ set M'})
  = (λa. {#lit-of a#}) '
    ∪ (set 'snd ' set (get-all-marked-decomposition M'))
    ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M'}
  by auto
have N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M'} ⊨ps
  (λa. {#lit-of a#}) ' (∪ (set 'snd ' set (get-all-marked-decomposition M'))
    ∪ {L | L. is-marked L ∧ L ∈ set M'})
  unfolding l using N by (auto simp add: all-in-true-clss-clss)
hence N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M'} ⊨ps (λa. {#lit-of a#}) ' set (tl Ls0)
  using M' unfolding LS LSM by auto
hence t: N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M'}
  ⊨ps (λa. {#lit-of a#}) ' set (tl Ls0)
  by (blast intro: all-in-true-clss-clss)
hence N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M}
  ⊨ps (λa. {#lit-of a#}) ' set (tl Ls0)
  using M'-in-M true-clss-clss-subset[OF - t,
    of N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M}]
  by auto
hence N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M} ⊨ps (λa. {#lit-of a#}) ' set Ls0
  using hd-Ls0 by (case-tac Ls0, auto)

moreover have (λa. {#lit-of a#}) ' set Ls0 ∪ N ⊨ps (λa. {#lit-of a#}) ' set seen0
  using Suc.premis unfolding Ls0 all-decomposition-implies-def by simp
moreover have ∧M Ma. (M::'a literal multiset set) ∪ Ma ⊨ps M
  by (simp add: all-in-true-clss-clss)
ultimately have Ψ: N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set M} ⊨ps
  (λa. {#lit-of a#}) ' set seen0
  by (meson true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r)
have (λa. {#lit-of a#}) '(set seen0
  ∪ (∪x∈set (get-all-marked-decomposition M'). set (snd x)))
  = (λa. {#lit-of a#}) ' set seen0
  ∪ (λa. {#lit-of a#}) ' (∪x∈set (get-all-marked-decomposition M'). set (snd x))
  by auto

hence ?case unfolding Ls0 using Ψ ΨN by simp
}
ultimately have ?case by auto
}
ultimately show ?case by arith
qed

```

**lemma** *all-decomposition-implies-propagated-lits-are-implied:*

**assumes** *all-decomposition-implies*  $N$  (*get-all-marked-decomposition*  $M$ )

**shows**  $N \cup \{\{\#lit-of\ L\# \mid L. is-marked\ L \wedge L \in set\ M\} \models_{ps} (\lambda a. \{\#lit-of\ a\#\}) \text{ ' set } M$   
*(is ?I  $\models_{ps}$  ?A)*

**proof** –

**have**  $?I \models_{ps} (\lambda a. \{\#lit-of\ a\#\}) \text{ ' } \{L \mid L. is-marked\ L \wedge L \in set\ M\}$

**by** (*auto intro: all-in-true-clss-clss*)

**moreover have**  $?I \models_{ps} (\lambda a. \{\#lit-of\ a\#\}) \text{ ' } \bigcup (set \text{ ' snd ' set (get-all-marked-decomposition } M))$

**using** *all-decomposition-implies-trail-is-implied* **assms** **by** *blast*

**ultimately have**  $N \cup \{\{\#lit-of\ m\# \mid m. is-marked\ m \wedge m \in set\ M\}$

$\models_{ps} (\lambda m. \{\#lit-of\ m\#\}) \text{ ' } \bigcup (set \text{ ' snd ' set (get-all-marked-decomposition } M))$

$\cup (\lambda m. \{\#lit-of\ m\#\}) \text{ ' } \{m \mid m. is-marked\ m \wedge m \in set\ M\}$

**by** *blast*

**thus** *?thesis*

**by** (*metis (no-types) get-all-marked-decomposition-snd-union[of M] image-Un*)

**qed**

**lemma** *all-decomposition-implies-insert-single:*

*all-decomposition-implies*  $N\ M \implies all-decomposition-implies (insert\ C\ N)\ M$

**unfolding** *all-decomposition-implies-def* **by** *auto*

### 13.4 Negation of Clauses

**definition**  $CNot :: 'v\ clause \Rightarrow 'v\ clauses$  **where**

$CNot\ \psi = \{ \{\#-L\# \mid L. L \in \# \psi \}$

**lemma** *in-CNot-uminus[iff]:*

**shows**  $\{\#L\# \in CNot\ \psi \longleftrightarrow -L \in \# \psi$

**using** *assms* **unfolding** *CNot-def* **by** *force*

**lemma** *CNot-singleton[simp]:*  $CNot\ \{\#L\# \} = \{\{\#-L\#\}\}$  **unfolding** *CNot-def* **by** *auto*

**lemma** *CNot-empty[simp]:*  $CNot\ \{\#\} = \{\}$  **unfolding** *CNot-def* **by** *auto*

**lemma** *CNot-plus[simp]:*  $CNot\ (A + B) = CNot\ A \cup CNot\ B$  **unfolding** *CNot-def* **by** *auto*

**lemma** *CNot-eq-empty[iff]:*

$CNot\ D = \{\} \longleftrightarrow D = \{\#\}$

**unfolding** *CNot-def* **by** (*auto simp add: multiset-eqI*)

**lemma** *in-CNot-implies-uminus:*

**assumes**  $L \in \# D$

**and**  $M \models_{as} CNot\ D$

**shows**  $M \models_a \{\#-L\#$  **and**  $-L \in lits-of\ M$

**using** *assms* **by** (*auto simp add: true-annots-def true-annot-def CNot-def*)

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

$CNot\ (remdups-mset\ A) = CNot\ A$

**unfolding** *CNot-def* **by** *auto*

**lemma** *Ball-CNot-Ball-mset[simp] :*

$(\forall x \in CNot\ D. P\ x) \longleftrightarrow (\forall L \in \# D. P\ \{\#-L\#)$

**unfolding** *CNot-def* **by** *auto*

**lemma** *consistent-CNot-not:*

**assumes** *consistent-interp*  $I$

**shows**  $I \models_s CNot\ \varphi \implies \neg I \models \varphi$

**using** *assms* **unfolding** *consistent-interp-def true-clss-def true-cls-def* **by** *auto*

**lemma** *total-not-true-cls-true-clss-CNot*:  
**assumes** *total-over-m I {φ}* **and**  $\neg I \models \varphi$   
**shows**  $I \models_s CNot \varphi$   
**using** *assms* **unfolding** *total-over-m-def total-over-set-def true-clss-def true-cls-def CNot-def*  
**apply** *clarify*  
**by** (case-tac *L*) (force intro: *pos-lit-in-atms-of neg-lit-in-atms-of*) +

**lemma** *total-not-CNot*:  
**assumes** *total-over-m I {φ}* **and**  $\neg I \models_s CNot \varphi$   
**shows**  $I \models \varphi$   
**using** *assms* *total-not-true-cls-true-clss-CNot* **by** *auto*

**lemma** *atms-of-m-CNot-atms-of[simp]*:  
*atms-of-m (CNot C) = atms-of C*  
**unfolding** *atms-of-m-def atms-of-def CNot-def* **by** *fastforce*

**lemma** *true-clss-clss-contradiction-true-clss-cls-false*:  
 $C \in D \implies D \models_{ps} CNot C \implies D \models_p \{\#\}$   
**unfolding** *true-clss-clss-def true-clss-cls-def total-over-m-def*  
**by** (metis *Un-commute atms-of-empty atms-of-m-CNot-atms-of atms-of-m-insert atms-of-m-union*  
*consistent-CNot-not insert-absorb sup-bot.left-neutral true-clss-def*)

**lemma** *true-annots-CNot-all-atms-defined*:  
**assumes**  $M \models_{as} CNot T$  **and**  $a1: L \in \# T$   
**shows** *atm-of L ∈ atm-of ‘ lits-of M*  
**by** (metis *assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton*)

**lemma** *true-clss-clss-false-left-right*:  
**assumes**  $\{\{\#L\#\}\} \cup B \models_p \{\#\}$   
**shows**  $B \models_{ps} CNot \{\#L\#\}$   
**unfolding** *true-clss-clss-def true-clss-cls-def*

**proof** (intro *allI impI*)  
**fix** *I*  
**assume**  
*tot: total-over-m I (B ∪ CNot {#L#}) and*  
*cons: consistent-interp I and*  
*I: I ⊨<sub>s</sub> B*  
**have** *total-over-m I ({#L#} ∪ B)* **using** *tot* **by** *auto*  
**hence**  $\neg I \models_s insert \{\#L\# \} B$   
**using** *assms cons* **unfolding** *true-clss-cls-def* **by** *simp*  
**thus**  $I \models_s CNot \{\#L\#\}$   
**using** *tot I* **by** (cases *L*) *auto*

**qed**

**lemma** *true-annots-true-cls-def-iff-negation-in-model*:  
 $M \models_{as} CNot C \iff (\forall L \in \# C. \neg L \in \text{lits-of } M)$   
**unfolding** *CNot-def true-annots-true-cls true-clss-def* **by** *auto*

**lemma** *consistent-CNot-not-tautology*:  
 $consistent-interp M \implies M \models_s CNot D \implies \neg \text{tautology } D$   
**by** (metis *atms-of-m-CNot-atms-of consistent-CNot-not satisfiable-carac' satisfiable-def*  
*tautology-def total-over-m-def*)

**lemma** *atms-of-m-CNot-atms-of-m*:  $\text{atms-of-m } (C\text{Not } CC) = \text{atms-of-m } \{CC\}$   
**by** *simp*

**lemma** *total-over-m-CNot-toal-over-m*[*simp*]:  
 $\text{total-over-m } I (C\text{Not } C) = \text{total-over-set } I (\text{atms-of } C)$   
**unfolding** *total-over-m-def total-over-set-def* **by** *auto*

**lemma** *uminus-lit-swap*:  $-(a::'a \text{ literal}) = i \longleftrightarrow a = -i$   
**by** *auto*

**lemma** *true-clss-clss-plus-CNot*:  
**assumes** *CC-L*:  $A \models_p CC + \{\#L\# \}$   
**and** *CNot-CC*:  $A \models_{ps} C\text{Not } CC$   
**shows**  $A \models_p \{\#L\# \}$   
**unfolding** *true-clss-clss-def true-clss-clss-def CNot-def total-over-m-def*

**proof** (*intro allI impI*)

**fix** *I*

**assume** *tot*:  $\text{total-over-set } I (\text{atms-of-m } (A \cup \{\{\#L\#\}\}))$

**and** *cons*: *consistent-interp* *I*

**and** *I*:  $I \models_s A$

**let**  $?I = I \cup \{Pos\ P | P. P \in \text{atms-of } CC \wedge P \notin \text{atm-of } 'I\}$

**have** *cons'*: *consistent-interp*  $?I$

**using** *cons* **unfolding** *consistent-interp-def*

**by** (*auto simp add: uminus-lit-swap atms-of-def rev-image-eqI*)

**have** *I'*:  $?I \models_s A$

**using** *I* *true-clss-union-increase* **by** *blast*

**have** *tot-CNot*:  $\text{total-over-m } ?I (A \cup C\text{Not } CC)$

**using** *tot* *atms-of-s-def* **by** (*fastforce simp add: total-over-m-def total-over-set-def*)

**hence** *tot-I-A-CC-L*:  $\text{total-over-m } ?I (A \cup \{CC + \{\#L\#\}\})$

**using** *tot* **unfolding** *total-over-m-def total-over-set-atm-of* **by** *auto*

**hence**  $?I \models CC + \{\#L\# \}$  **using** *CC-L cons' I'* **unfolding** *true-clss-clss-def* **by** *blast*

**moreover**

**have**  $?I \models_s C\text{Not } CC$  **using** *CNot-CC cons' I'* *tot-CNot* **unfolding** *true-clss-clss-def* **by** *auto*

**hence**  $\neg A \models_p CC$

**by** (*metis* (*no-types, lifting*) *I'* *atms-of-m-CNot-atms-of-m atms-of-m-union cons'*  
*consistent-CNot-not tot-CNot total-over-m-def true-clss-clss-def*)

**hence**  $\neg ?I \models CC$  **using**  $\langle ?I \models_s C\text{Not } CC \rangle$  *cons'* *consistent-CNot-not* **by** *blast*

**ultimately have**  $?I \models \{\#L\# \}$  **by** *blast*

**thus**  $I \models \{\#L\# \}$

**by** (*metis* (*no-types, lifting*) *atms-of-m-union cons'* *consistent-CNot-not tot total-not-CNot*  
*total-over-m-def total-over-set-union true-clss-union-increase*)

**qed**

**lemma** *true-annots-CNot-lit-of-notin-skip*:

**assumes** *LM*:  $L \# M \models_{as} C\text{Not } A$  **and** *LA*:  $\text{lit-of } L \notin \# A - \text{lit-of } L \notin \# A$

**shows**  $M \models_{as} C\text{Not } A$

**using** *LM* **unfolding** *true-annots-def Ball-def*

**proof** (*intro allI impI*)

**fix** *l*

**assume** *H*:  $\forall x. x \in C\text{Not } A \longrightarrow L \# M \models_a x$  **and** *l*:  $l \in C\text{Not } A$

**hence**  $L \# M \models_a l$  **by** *auto*

**thus**  $M \models_a l$  **using** *LA l* **by** (*cases L*) (*auto simp add: CNot-def*)

**qed**

**lemma** *true-clss-clss-union-false-true-clss-clss-cnot*:  
 $A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} CNot\ B$   
**using** *total-not-CNot consistent-CNot-not unfolding total-over-m-def true-clss-clss-def*  
**by** *fastforce*

**lemma** *true-annot-remove-hd-if-notin-vars*:  
**assumes**  $a \# M' \models_a D$   
**and**  $atm\text{-}of\ (lit\text{-}of\ a) \notin atm\text{-}of\ D$   
**shows**  $M' \models_a D$   
**using** *assms true-clss-remove-hd-if-notin-vars unfolding true-annot-def* **by** *auto*

**lemma** *true-annot-remove-if-notin-vars*:  
**assumes**  $M @ M' \models_a D$   
**and**  $\forall x \in atm\text{-}of\ D. x \notin atm\text{-}of\ 'lits\text{-}of\ M$   
**shows**  $M' \models_a D$   
**using** *assms apply (induct M, simp)*  
**using** *true-annot-remove-hd-if-notin-vars* **by** *force+*

**lemma** *true-annots-remove-if-notin-vars*:  
**assumes**  $M @ M' \models_{as} D$   
**and**  $\forall x \in atm\text{-}of\text{-}m\ D. x \notin atm\text{-}of\ 'lits\text{-}of\ M$   
**shows**  $M' \models_{as} D$  **unfolding** *true-annots-def*  
**using** *assms true-annot-remove-if-notin-vars[of M M']*  
**unfolding** *true-annots-def atm\text{-}of\text{-}m\text{-}def* **by** *force*

**lemma** *all-variables-defined-not-imply-cnot*:  
**assumes**  $\forall s \in atm\text{-}of\text{-}m\ \{B\}. s \in atm\text{-}of\ 'lits\text{-}of\ A$   
**and**  $\neg A \models_a B$   
**shows**  $A \models_{as} CNot\ B$   
**unfolding** *true-annot-def true-annots-def Ball-def CNot-def true-lit-def*

**proof** (*clarify, rule ccontr*)  
**fix**  $L$   
**assume**  $LB: L \in \# B$  **and**  $\neg lits\text{-}of\ A \models_l - L$   
**hence**  $atm\text{-}of\ L \in atm\text{-}of\ 'lits\text{-}of\ A$   
**using** *assms(1) by (simp add: atm\text{-}of\ lit\text{-}in\ atm\text{-}of\ lits\text{-}of\ def)*  
**hence**  $L \in lits\text{-}of\ A \vee -L \in lits\text{-}of\ A$   
**using** *atm\text{-}of\ in\ atm\text{-}of\ set\ iff\ in\ set\ or\ uminus\ in\ set* **by** *metis*  
**hence**  $L \in lits\text{-}of\ A$  **using**  $(\neg lits\text{-}of\ A \models_l - L)$  **by** *auto*  
**thus** *False*  
**using**  $LB$  *assms(2) unfolding true-annot-def true-lit-def true-clss-def Bex-mset-def*  
**by** *blast*  
**qed**

**lemma** *CNot-union-mset[simp]*:  
 $CNot\ (A \# \cup B) = CNot\ A \cup CNot\ B$   
**unfolding** *CNot-def* **by** *auto*

## 13.5 Other

**abbreviation**  $no\text{-}dup\ L \equiv distinct\ (map\ (\lambda l. atm\text{-}of\ (lit\text{-}of\ l))\ L)$

**lemma** *no-dup-rev[simp]*:  
 $no\text{-}dup\ (rev\ M) \longleftrightarrow no\text{-}dup\ M$   
**by** (*auto simp: rev-map[symmetric]*)

**lemma** *no-dup-length-eq-card-atm-of-lits-of*:

**assumes** *no-dup M*  
**shows** *length M = card (atm-of ‘ lits-of M)*  
**using** *assms unfolding lits-of-def by (induct M) (auto simp add: image-image)*

**lemma** *distinctconsistent-interp:*

*no-dup M  $\implies$  consistent-interp (lits-of M)*

**proof** *(induct M)*

**case** *Nil*

**show** *?case by auto*

**next**

**case** *(Cons L M)*

**hence** *a1: consistent-interp (lits-of M) by auto*

**have** *a2: atm-of (lit-of L)  $\notin$  ( $\lambda l.$  atm-of (lit-of l)) ‘ set M using Cons.prem by auto*

**have** *undefined-lit M (lit-of L)*

**using** *a2 image-iff unfolding defined-lit-def by fastforce*

**thus** *?case*

**using** *a1 by simp*

**qed**

**lemma** *distinctget-all-marked-decomposition-no-dup:*

**assumes** *(a, b)  $\in$  set (get-all-marked-decomposition M)*

**and** *no-dup M*

**shows** *no-dup (a @ b)*

**using** *assms by force*

**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*

**proof** *–*

**have**  $\forall l \in \# A. \neg l \in \text{lits-of } (L \# M)$

**using** *assms(1) in-CNot-implies-uminus(2) by blast*

**moreover**

**have** *atm-of (lit-of L)  $\notin$  atm-of ‘ lits-of M*

**using** *assms(3) lits-of-def by force*

**hence**  *$\neg$  lit-of L  $\notin$  lits-of M unfolding lits-of-def*

**by** *(metis (no-types) atm-of-uminus imageI)*

**ultimately have**  $\forall l \in \# A. \neg l \in \text{lits-of } M$

**using** *assms(2) unfolding Ball-mset-def by (metis insertE lits-of-cons uminus-of-uminus-id)*

**thus** *?thesis by (auto simp add: true-annots-def)*

**qed**

**type-synonym** *'v clauses = 'v clause multiset*

**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:: 'a clauses  $\Rightarrow$  'a clauses  $\Rightarrow$  bool (infix  $\models_{psm}$  50) where*

*I  $\models_{psm}$  C  $\equiv$  set-mset I  $\models_{ps}$  (set-mset C)*

Analog of  $\llbracket ?N \models_{ps} ?B; ?A \subseteq ?B \rrbracket \implies ?N \models_{ps} ?A$

**lemma** *true-clss-clssm-subsetE: N  $\models_{psm}$  B  $\implies$  A  $\subseteq \#$  B  $\implies$  N  $\models_{psm}$  A*

**using** *set-mset-mono true-clss-clss-subsetE by blast*

**abbreviation** *true-clss-cls-m*:: 'a clauses  $\Rightarrow$  'a clause  $\Rightarrow$  bool (**infix**  $\models_{pm}$  50) **where**  
 $I \models_{pm} C \equiv \text{set-mset } I \models_p C$

**abbreviation** *distinct-mset-mset* :: 'a multiset multiset  $\Rightarrow$  bool **where**  
 $\text{distinct-mset-mset } \Sigma \equiv \text{distinct-mset-set } (\text{set-mset } \Sigma)$

**abbreviation** *all-decomposition-implies-m* **where**  
 $\text{all-decomposition-implies-m } A \ B \equiv \text{all-decomposition-implies } (\text{set-mset } A) \ B$

**abbreviation** *atms-of-mu* **where**  
 $\text{atms-of-mu } U \equiv \text{atms-of-m } (\text{set-mset } U)$

**abbreviation** *true-clss-m*:: 'a interp  $\Rightarrow$  'a clauses  $\Rightarrow$  bool (**infix**  $\models_{sm}$  50) **where**  
 $I \models_{sm} C \equiv I \models_s \text{set-mset } C$

**abbreviation** *true-clss-ext-m* (**infix**  $\models_{sextm}$  49) **where**  
 $I \models_{sextm} C \equiv I \models_{sext} \text{set-mset } C$

**end**

**theory** *CDCL-NOT*

**imports** *Partial-Annotated-Clausal-Logic List-More Wellfounded-More Partial-Clausal-Logic*  
**begin**

## 14 NOT's CDCL

**sledgehammer-params**[*verbose, prover=e spass z3 cvc4 verit remote-vampire*]

**declare** *set-mset-minus-replicate-mset*[*simp*]

### 14.1 Auxiliary Lemmas and Measure

**lemma** *no-dup-cannot-not-lit-and-uminus*:

$\text{no-dup } M \Longrightarrow \neg \text{lit-of } xa = \text{lit-of } x \Longrightarrow x \in \text{set } M \Longrightarrow xa \notin \text{set } M$   
**by** (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id*)

**lemma** *true-clss-single-iff-incl*:

$I \models_s \text{single } 'B \longleftrightarrow B \subseteq I$   
**unfolding** *true-clss-def* **by** *auto*

**lemma** *atms-of-m-single-atm-of*[*simp*]:

$\text{atms-of-m } \{\{\# \text{lit-of } L \# \} \mid L. P \ L\} = \text{atm-of } ' \{\text{lit-of } L \mid L. P \ L\}$   
**unfolding** *atms-of-m-def* **by** *auto*

**lemma** *atms-of-uminus-lit-atm-of-lit-of*:

$\text{atms-of } \{\# \neg \text{lit-of } x. x \in \# \ A \# \} = \text{atm-of } ' (\text{lit-of } ' (\text{set-mset } A))$   
**unfolding** *atms-of-def* **by** (*auto simp add: Fun.image-comp*)

**lemma** *atms-of-m-single-image-atm-of-lit-of*:

$\text{atms-of-m } ((\lambda x. \{\# \text{lit-of } x \# \}) ' A) = \text{atm-of } ' (\text{lit-of } ' A)$   
**unfolding** *atms-of-m-def* **by** *auto*

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**  $\mu_C\text{-nil}[simp]$ :

$\mu_C\ s\ b\ [] = 0$

**unfolding**  $\mu_C\text{-def}$  **by** *auto*

**lemma**  $\mu_C\text{-single}[simp]$ :

$\mu_C\ s\ b\ [L] = L * b \wedge (s - \text{Suc } 0)$

**unfolding**  $\mu_C\text{-def}$  **by** *auto*

**lemma** *set-sum-atLeastLessThan-add*:

$(\sum i=k..<k+(b::nat). f\ i) = (\sum i=0..<b. f\ (k + i))$

**by** (*induction b*) *auto*

**lemma** *set-sum-atLeastLessThan-Suc*:

$(\sum i=1..<\text{Suc } j. f\ i) = (\sum i=0..<j. f\ (\text{Suc } i))$

**using** *set-sum-atLeastLessThan-add*[*of - 1 j*] **by** *force*

**lemma**  $\mu_C\text{-cons}$ :

$\mu_C\ s\ b\ (L \# M) = L * b \wedge (s - 1 - \text{length } M) + \mu_C\ s\ b\ M$

**proof** –

**have**  $\mu_C\ s\ b\ (L \# M) = (\sum i=0..<\text{length } (L\#M). (L\#M)!i * b \wedge (s + i - \text{length } (L\#M)))$

**unfolding**  $\mu_C\text{-def}$  **by** *blast*

**also have**  $\dots = (\sum i=0..<1. (L\#M)!i * b \wedge (s + i - \text{length } (L\#M)))$

$+ (\sum i=1..<\text{length } (L\#M). (L\#M)!i * b \wedge (s + i - \text{length } (L\#M)))$

**by** (*rule setsum-add-nat-ivl[symmetric]*) *simp-all*

**finally have**  $\mu_C\ s\ b\ (L \# M) = L * b \wedge (s - 1 - \text{length } M)$

$+ (\sum i=1..<\text{length } (L\#M). (L\#M)!i * b \wedge (s + i - \text{length } (L\#M)))$

**by** *auto*

**moreover** {

**have**  $(\sum i=1..<\text{length } (L\#M). (L\#M)!i * b \wedge (s + i - \text{length } (L\#M))) =$

$(\sum i=0..<\text{length } (M). (L\#M)!(\text{Suc } i) * b \wedge (s + (\text{Suc } i) - \text{length } (L\#M)))$

**unfolding** *length-Cons set-sum-atLeastLessThan-Suc* **by** *blast*

**also have**  $\dots = (\sum i=0..<\text{length } (M). M!i * b \wedge (s + i - \text{length } M))$

**by** *auto*

**finally have**  $(\sum i=1..<\text{length } (L\#M). (L\#M)!i * b \wedge (s + i - \text{length } (L\#M))) = \mu_C\ s\ b\ M$

**unfolding**  $\mu_C\text{-def}$  .

}

**ultimately show** *?thesis* **by** *presburger*

**qed**

**lemma**  $\mu_C\text{-append}$ :

**assumes**  $s \geq \text{length } (M @ M')$

**shows**  $\mu_C\ s\ b\ (M @ M') = \mu_C\ (s - \text{length } M')\ b\ M + \mu_C\ s\ b\ M'$

**proof** –

**have**  $\mu_C\ s\ b\ (M @ M') = (\sum i=0..<\text{length } (M @ M'). (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$

**unfolding**  $\mu_C\text{-def}$  **by** *blast*

**moreover then have**  $\dots = (\sum i=0..<\text{length } M. (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$

$+ (\sum i=\text{length } M..<\text{length } (M @ M'). (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$

**by** (*auto intro!: setsum-add-nat-ivl[symmetric]*)

**moreover**

**have**  $\forall i \in \{0..<\text{length } M\}. (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')) = M!i * b \wedge (s - \text{length } M' + i - \text{length } M)$

**using**  $\langle s \geq \text{length } (M @ M') \rangle$  **by** (*auto simp add: nth-append ac-simps*)

**then have**  $\mu_C\ (s - \text{length } M')\ b\ M = (\sum i=0..<\text{length } M. (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$



unfolding  $\mu_C$ -def by auto  
 ultimately have  $\mu_C \ s \ b \ (M @ M') = \mu_C \ (s - \text{length } M') \ b \ M$   
 $\quad + (\sum_{i=\text{length } M}..<\text{length } (M @ M')). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M'))$   
 by auto  
 moreover {  
 have  $(\sum_{i=\text{length } M}..<\text{length } (M @ M')). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')) =$   
 $(\sum_{i=0}..<\text{length } M'. M!i * b^\wedge (s + i - \text{length } M'))$   
 unfolding length-append set-sum-atLeastLessThan-add by auto  
 then have  $(\sum_{i=\text{length } M}..<\text{length } (M @ M')). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')) = \mu_C \ s \ b$   
 $M'$   
 unfolding  $\mu_C$ -def .  
 }  
 ultimately show ?thesis by presburger  
 qed

lemma  $\mu_C$ -cons-non-empty-inf:  
 assumes  $M\text{-ge-1}: \forall i \in \text{set } M. i \geq 1$  and  $M: M \neq []$   
 shows  $\mu_C \ s \ b \ M \geq b^\wedge (s - \text{length } M)$   
 using assms by (cases M) (auto simp: mult-eq-if  $\mu_C$ -cons)

Duplicate of " /src/HOL/ex/NatSum.thy" (but generalized to  $(0::'a) \leq k$ )

lemma sum-of-powers:  $0 \leq k \implies (k - 1) * (\sum_{i=0}..<n. k^\wedge i) = k^\wedge n - (1::nat)$   
 apply (cases k = 0)  
 apply (cases n; simp)  
 by (induct n) (auto simp: Nat.nat-distrib)

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  
 $M\text{-le}: \forall i < \text{length } M. M!i < b$  and  
 $s \geq \text{length } M$   
 shows  $\mu_C \ s \ b \ M < b^\wedge s$   
 proof -  
 consider (b1)  $b = 1 \mid (b) \ b > 1$  using  $\langle b > 0 \rangle$  by (cases b) auto  
 then show ?thesis  
 proof cases  
 case b1  
 then have  $\forall i < \text{length } M. M!i = 0$  using M-le by auto  
 then have  $\mu_C \ s \ b \ M = 0$  unfolding  $\mu_C$ -def by auto  
 then show ?thesis using  $\langle b > 0 \rangle$  by auto  
 next  
 case b  
 have  $\forall i \in \{0..<\text{length } M\}. M!i * b^\wedge (s + i - \text{length } M) \leq (b-1) * b^\wedge (s + i - \text{length } M)$   
 using M-le  $\langle b > 1 \rangle$  by auto  
 then have  $\mu_C \ s \ b \ M \leq (\sum_{i=0}..<\text{length } M. (b-1) * b^\wedge (s + i - \text{length } M))$   
 using  $\langle M \neq [] \rangle \langle b > 0 \rangle$  unfolding  $\mu_C$ -def by (auto intro: setsum-mono)  
 also  
 have  $\forall i \in \{0..<\text{length } M\}. (b-1) * b^\wedge (s + i - \text{length } M) = (b-1) * b^\wedge i * b^\wedge (s - \text{length } M)$   
 by (metis Nat.add-diff-assoc2 add.commute assms(4) mult.assoc power-add)  
 then have  $(\sum_{i=0}..<\text{length } M. (b-1) * b^\wedge (s + i - \text{length } M))$   
 $= (\sum_{i=0}..<\text{length } M. (b-1) * b^\wedge i * b^\wedge (s - \text{length } M))$ 
end

```

    by (auto simp add: ac-simps)
  also have ... = (∑ i=0.. $\text{length } M$ .  $b^i$ ) *  $b^{(s - \text{length } M)}$  *  $(b-1)$ 
    by (simp add: setsum-left-distrib setsum-right-distrib ac-simps)
  finally have  $\mu_C s b M \leq (\sum i=0.. $\text{length } M$ .  $b^i$ ) *  $(b-1)$  *  $b^{(s - \text{length } M)}$ 
    by (simp add: ac-simps)

  also
    have (∑ i=0.. $\text{length } M$ .  $b^i$ ) *  $(b-1)$  =  $b^{\text{length } M - 1}$ 
      using sum-of-powers[of  $b$   $\text{length } M$ ]  $\langle b > 1 \rangle$ 
      by (auto simp add: ac-simps)
    finally have  $\mu_C s b M \leq (b^{\text{length } M - 1}) * b^{(s - \text{length } M)}$ 
      by auto
    also have ... <  $b^{\text{length } M} * b^{(s - \text{length } M)}$ 
      using  $\langle b > 1 \rangle$  by auto
    also have ... =  $b^s$ 
      by (metis assms(4) le-add-diff-inverse power-add)
    finally show ?thesis unfolding  $\mu_C$ -def by (auto simp add: ac-simps)
qed
qed$ 
```

In the degenerate case  $b = (0::'a)$ , the list  $M$  is empty (since the list cannot contain any element).

```

lemma  $\mu_C$ -bounded:
  fixes  $b :: \text{nat}$ 
  assumes
     $M$ -le:  $\forall i < \text{length } M. M!i < b$  and
     $s \geq \text{length } M$ 
     $b > 0$ 
  shows  $\mu_C s b M < b^s$ 
proof -
  consider ( $M0$ )  $M = []$  | ( $M$ )  $b > 0$  and  $M \neq []$ 
  using  $M$ -le by (cases  $b$ , cases  $M$ ) auto
  then show ?thesis
  proof cases
    case  $M0$ 
    then show ?thesis using  $M$ -le  $\langle b > 0 \rangle$  by auto
  next
    case  $M$ 
    show ?thesis using  $\mu_C$ -bounded-non-degenerated[OF  $M$  assms(1,2)] by arith
  qed
qed
qed

```

When  $b = 0$ , we cannot show that the measure is empty, since  $0^0 = 1$ .

```

lemma  $\mu_C$ -base-0:
  assumes  $\text{length } M \leq s$ 
  shows  $\mu_C s 0 M \leq M!0$ 
proof -
  {
    assume  $s = \text{length } M$ 
    moreover {
      fix  $n$ 
      have (∑ i=0.. $n$ .  $M!i * (0::\text{nat})^i$ ) ≤  $M!0$ 
        apply (induction  $n$  rule: nat-induct)
        by simp (case-tac  $n$ , auto)
    }
  }

```

```

    ultimately have ?thesis unfolding  $\mu_C$ -def by auto
  }
  moreover
  {
    assume length  $M < s$ 
    then have  $\mu_C \ s \ 0 \ M = 0$  unfolding  $\mu_C$ -def by auto
    ultimately show ?thesis using assms unfolding  $\mu_C$ -def by linarith
  }
qed

```

## 14.2 Initial definitions

### 14.2.1 The state

We define here an abstraction over operation on the state we are manipulating.

```

locale dpll-state =
  fixes
    trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
    clauses :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
    tl-trail :: 'st  $\Rightarrow$  'st and
    add-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    remove-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
  assumes
    trail-prepend-trail[simp]:
       $\bigwedge st \ L. \text{undefined-lit } (trail \ st) \ (\text{lit-of } L) \Longrightarrow trail \ (\text{prepend-trail } L \ st) = L \ \# \ trail \ st$ 
and
    tl-trail[simp]: trail (tl-trail  $S$ ) = tl (trail  $S$ ) and
    trail-add-clsNOT[simp]:  $\bigwedge st \ C. \text{no-dup } (trail \ st) \Longrightarrow trail \ (\text{add-cls}_{NOT} \ C \ st) = trail \ st$  and
    trail-remove-clsNOT[simp]:  $\bigwedge st \ C. trail \ (\text{remove-cls}_{NOT} \ C \ st) = trail \ st$  and

    clauses-prepend-trail[simp]:
       $\bigwedge st \ L. \text{undefined-lit } (trail \ st) \ (\text{lit-of } L) \Longrightarrow clauses \ (\text{prepend-trail } L \ st) = clauses \ st$ 
and
    clauses-tl-trail[simp]:  $\bigwedge st. clauses \ (tl-trail \ st) = clauses \ st$  and
    clauses-add-clsNOT[simp]:
       $\bigwedge st \ C. \text{no-dup } (trail \ st) \Longrightarrow clauses \ (\text{add-cls}_{NOT} \ C \ st) = \{\#C\# \} + clauses \ st$  and
    clauses-remove-clsNOT[simp]:  $\bigwedge st \ C. clauses \ (\text{remove-cls}_{NOT} \ C \ st) = \text{remove-mset } C \ (clauses \ st)$ 
  begin

  function reduce-trail-toNOT :: ('v, unit, unit) marked-lits  $\Rightarrow$  'st  $\Rightarrow$  'st where
    reduce-trail-toNOT  $F \ S =$ 
      (if length (trail  $S$ ) = length  $F \vee trail \ S = []$  then  $S$  else reduce-trail-toNOT  $F \ (tl-trail \ S))$ 
  by fast+
  termination by (relation measure ( $\lambda(-, S). \text{length } (trail \ S)$ )) auto
  declare reduce-trail-toNOT.simps[simp del]

```

**lemma**

**shows**

```

  reduce-trail-toNOT-nil[simp]: trail  $S = [] \Longrightarrow reduce-trail-to_{NOT} \ F \ S = S$  and
  reduce-trail-toNOT-eq-length[simp]: length (trail  $S$ ) = length  $F \Longrightarrow reduce-trail-to_{NOT} \ F \ S = S$ 
by (auto simp: reduce-trail-toNOT.simps)

```

**lemma** reduce-trail-to<sub>NOT</sub>-length-ne[simp]:

```

  length (trail  $S$ )  $\neq$  length  $F \Longrightarrow trail \ S \neq [] \Longrightarrow$ 
    reduce-trail-toNOT  $F \ S = reduce-trail-to_{NOT} \ F \ (tl-trail \ S)$ 

```

**by** (*auto simp: reduce-trail-to<sub>NOT</sub>.simps*)

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-length-le*:

**assumes** *length F > length (trail S)*

**shows** *trail (reduce-trail-to<sub>NOT</sub> F S) = []*

**using** *assms by (induction F S rule: reduce-trail-to<sub>NOT</sub>.induct)*

*(simp add: less-imp-diff-less reduce-trail-to<sub>NOT</sub>.simps)*

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-nil[simp]*:

*trail (reduce-trail-to<sub>NOT</sub> [] S) = []*

**by** (*induction []:: ('v, unit, unit) marked-lits S rule: reduce-trail-to<sub>NOT</sub>.induct*)

*(simp add: less-imp-diff-less reduce-trail-to<sub>NOT</sub>.simps)*

**lemma** *clauses-reduce-trail-to<sub>NOT</sub>-nil*:

*clauses (reduce-trail-to<sub>NOT</sub> [] S) = clauses S*

**by** (*induction []:: ('v, unit, unit) marked-lits S rule: reduce-trail-to<sub>NOT</sub>.induct*)

*(simp add: less-imp-diff-less reduce-trail-to<sub>NOT</sub>.simps)*

**lemma** *reduce-trail-to<sub>NOT</sub>-skip-beginning*:

**assumes** *trail S = F' @ F*

**shows** *trail (reduce-trail-to<sub>NOT</sub> F S) = F*

**using** *assms by (induction F' arbitrary: S) auto*

**lemma** *reduce-trail-to<sub>NOT</sub>-clauses[simp]*:

*clauses (reduce-trail-to<sub>NOT</sub> F S) = clauses S*

**by** (*induction F S rule: reduce-trail-to<sub>NOT</sub>.induct*)

*(simp add: less-imp-diff-less reduce-trail-to<sub>NOT</sub>.simps)*

**abbreviation** *trail-weight where*

*trail-weight S ≡ map ((λl. 1 + length l) o snd) (get-all-marked-decomposition (trail S))*

**definition** *state-eq<sub>NOT</sub> :: 'st ⇒ 'st ⇒ bool (infix ~ 50) where*

*S ~ T ⟷ trail S = trail T ∧ clauses S = clauses T*

**lemma** *state-eq<sub>NOT</sub>-ref[simp]*:

*S ~ S*

**unfolding** *state-eq<sub>NOT</sub>-def by auto*

**lemma** *state-eq<sub>NOT</sub>-sym*:

*S ~ T ⟷ T ~ S*

**unfolding** *state-eq<sub>NOT</sub>-def by auto*

**lemma** *state-eq<sub>NOT</sub>-trans*:

*S ~ T ⟹ T ~ U ⟹ S ~ U*

**unfolding** *state-eq<sub>NOT</sub>-def by auto*

**lemma**

**shows**

*state-eq<sub>NOT</sub>-trail: S ~ T ⟹ trail S = trail T and*

*state-eq<sub>NOT</sub>-clauses: S ~ T ⟹ clauses S = clauses T*

**unfolding** *state-eq<sub>NOT</sub>-def by auto*

**lemmas** *state-simp<sub>NOT</sub>[simp] = state-eq<sub>NOT</sub>-trail state-eq<sub>NOT</sub>-clauses*

**lemma** *trail-eq-reduce-trail-to<sub>NOT</sub>-eq*:

$trail\ S = trail\ T \implies trail\ (reduce\_trail\_to_{NOT}\ F\ S) = trail\ (reduce\_trail\_to_{NOT}\ F\ T)$   
**apply** (induction  $F\ S$  arbitrary:  $T$  rule:  $reduce\_trail\_to_{NOT}.induct$ )  
**by** (metis  $tl\_trail\ reduce\_trail\_to_{NOT}\ eq\_length\ reduce\_trail\_to_{NOT}\ length\_ne\ reduce\_trail\_to_{NOT}\ nil$ )

**lemma**  $reduce\_trail\_to_{NOT}\ state\_eq_{NOT}\ compatible$ :

**assumes**  $ST$ :  $S \sim T$

**shows**  $reduce\_trail\_to_{NOT}\ F\ S \sim reduce\_trail\_to_{NOT}\ F\ T$

**proof** –

**have**  $clauses(reduce\_trail\_to_{NOT}\ F\ S) = clauses(reduce\_trail\_to_{NOT}\ F\ T)$

**using**  $ST$  **by** *auto*

**moreover have**  $trail(reduce\_trail\_to_{NOT}\ F\ S) = trail(reduce\_trail\_to_{NOT}\ F\ T)$

**using**  $trail\_eq\_reduce\_trail\_to_{NOT}\ eq[of\ S\ T\ F]$   $ST$  **by** *auto*

**ultimately show** *?thesis* **by** (auto simp del:  $state\_simp_{NOT}$  simp:  $state\_eq_{NOT}\ def$ )

**qed**

**lemma**  $trail\_reduce\_trail\_to_{NOT}\ add\_cls_{NOT}[simp]$ :

*no-dup* ( $trail\ S$ )  $\implies$

$trail(reduce\_trail\_to_{NOT}\ F\ (add\_cls_{NOT}\ C\ S)) = trail(reduce\_trail\_to_{NOT}\ F\ S)$

**by** (rule  $trail\_eq\_reduce\_trail\_to_{NOT}\ eq$ ) *simp*

**lemma**  $reduce\_trail\_to_{NOT}\ trail\_tl\_trail\_decomp[simp]$ :

$trail\ S = F' @ Marked\ K\ () \# F \implies$

$(trail(reduce\_trail\_to_{NOT}\ F\ (tl\_trail\ S))) = F$

**apply** (rule  $reduce\_trail\_to_{NOT}\ skip\_beginning[of\ -\ tl\ (F' @ Marked\ K\ () \# [])]$ )

**by** (cases  $F'$ ) (auto simp add:  $tl\_append\ reduce\_trail\_to_{NOT}\ skip\_beginning$ )

**end**

## 14.2.2 Definition of the operation

**locale**  $propagate\_ops =$

*dpll-state*  $trail\ clauses\ prepend\_trail\ tl\_trail\ add\_cls_{NOT}\ remove\_cls_{NOT}$  **for**

$trail :: 'st \Rightarrow ('v, unit, unit)\ marked\_lits$  **and**

$clauses :: 'st \Rightarrow 'v\ clauses$  **and**

$prepend\_trail :: ('v, unit, unit)\ marked\_lit \Rightarrow 'st \Rightarrow 'st$  **and**

$tl\_trail :: 'st \Rightarrow 'st$  **and**

$add\_cls_{NOT}\ remove\_cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$  **and**

$propagate\_cond :: ('v, unit, unit)\ marked\_lit \Rightarrow 'st \Rightarrow bool$

**begin**

**inductive**  $propagate_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$  **where**

$propagate_{NOT}[intro]: C + \{\#L\# \} \in \# clauses\ S \implies trail\ S \models_{as}\ CNot\ C$

$\implies undefined\_lit\ (trail\ S)\ L$

$\implies propagate\_cond\ (Propagated\ L\ ())\ S$

$\implies T \sim prepend\_trail\ (Propagated\ L\ ())\ S$

$\implies propagate_{NOT}\ S\ T$

**inductive-cases**  $propagateE[elim]: propagate_{NOT}\ S\ T$

**end**

**locale**  $decide\_ops =$

*dpll-state*  $trail\ clauses\ prepend\_trail\ tl\_trail\ add\_cls_{NOT}\ remove\_cls_{NOT}$  **for**

$trail :: 'st \Rightarrow ('v, unit, unit)\ marked\_lits$  **and**

$clauses :: 'st \Rightarrow 'v\ clauses$  **and**

$prepend\_trail :: ('v, unit, unit)\ marked\_lit \Rightarrow 'st \Rightarrow 'st$  **and**

$tl\_trail :: 'st \Rightarrow 'st$  **and**

$add\_cls_{NOT}\ remove\_cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$

```

begin
inductive decideNOT :: 'st ⇒ 'st ⇒ bool where
decideNOT[intro]: undefined-lit (trail S) L ⇒ atm-of L ∈ atms-of-mu (clauses S)
  ⇒ T ∼ prepend-trail (Marked L ()) S
  ⇒ decideNOT S T

inductive-cases decideE[elim]: decideNOT S S'
end

locale backjumping-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT remove-clsNOT:: 'v clause ⇒ 'st ⇒ 'st +
fixes
  backjump-conds :: 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool
begin
inductive backjump where
trail S = F' @ Marked K () # F
  ⇒ T ∼ prepend-trail (Propagated L ()) (reduce-trail-toNOT F S)
  ⇒ C ∈ # clauses S
  ⇒ trail S ⊨as CNot C
  ⇒ undefined-lit F L
  ⇒ atm-of L ∈ atms-of-mu (clauses S) ∪ atm-of ' (lits-of (trail S))
  ⇒ clauses S ⊨pm C' + {#L#}
  ⇒ F ⊨as CNot C'
  ⇒ backjump-conds C' L S T
  ⇒ backjump S T
inductive-cases backjumpE: backjump S T
end

```

### 14.3 DPLL with backjumping

```

locale dpll-with-backjumping-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT +
  propagate-ops trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT propagate-conds +
  decide-ops trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT +
  backjumping-ops trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT backjump-conds
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT remove-clsNOT:: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool and
  backjump-conds :: 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool +
assumes
  bj-can-jump:
  ∧ S C F' K F L.
  inv S ⇒
  no-dup (trail S) ⇒
  trail S = F' @ Marked K () # F ⇒

```

$C \in \# \text{ clauses } S \implies$   
 $\text{trail } S \models_{as} C \text{Not } C \implies$   
 $\text{undefined-lit } F L \implies$   
 $\text{atm-of } L \in \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (F' @ \text{Marked } K () \# F)) \implies$   
 $\text{clauses } S \models_{pm} C' + \{\#L\# \} \implies$   
 $F \models_{as} C \text{Not } C' \implies$   
 $\neg \text{no-step backjump } S$

**begin**

We cannot add a like condition  $\text{atms-of } C' \subseteq \text{atms-of-m } N$  because to ensure that we can backjump even if the last decision variable has disappeared.

The part of the condition  $\text{atm-of } L \in \text{atm-of } ' (\text{lits-of } (F' @ \text{Marked } K () \# F))$  is important, otherwise you are not sure that you can backtrack.

### 14.3.1 Definition

We define dpll with backjumping:

**inductive**  $\text{dpll-bj} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **for**  $S :: 'st$  **where**

$\text{bj-decide}_{NOT} : \text{decide}_{NOT} S S' \implies \text{dpll-bj } S S' \mid$

$\text{bj-propagate}_{NOT} : \text{propagate}_{NOT} S S' \implies \text{dpll-bj } S S' \mid$

$\text{bj-backjump} : \text{backjump } S S' \implies \text{dpll-bj } S S'$

**lemmas**  $\text{dpll-bj-induct} = \text{dpll-bj.induct}[\text{split-format}(\text{complete})]$

**thm**  $\text{dpll-bj-induct}[\text{OF } \text{dpll-with-backjumping-ops-axioms}]$

**lemma**  $\text{dpll-bj-all-induct}[\text{consumes } 2, \text{case-names } \text{decide}_{NOT} \text{ propagate}_{NOT} \text{ backjump}] :$

**fixes**  $S T :: 'st$

**assumes**

$\text{dpll-bj } S T$  **and**

$\text{inv } S$

$\bigwedge L T. \text{undefined-lit } (\text{trail } S) L \implies \text{atm-of } L \in \text{atms-of-mu } (\text{clauses } S)$

$\implies T \sim \text{prepend-trail } (\text{Marked } L ()) S$

$\implies P S T$  **and**

$\bigwedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \implies \text{trail } S \models_{as} C \text{Not } C \implies \text{undefined-lit } (\text{trail } S) L$

$\implies T \sim \text{prepend-trail } (\text{Propagated } L ()) S$

$\implies P S T$  **and**

$\bigwedge C F' K F L C' T. C \in \# \text{ clauses } S \implies F' @ \text{Marked } K () \# F \models_{as} C \text{Not } C$

$\implies \text{trail } S = F' @ \text{Marked } K () \# F$

$\implies \text{undefined-lit } F L$

$\implies \text{atm-of } L \in \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (F' @ \text{Marked } K () \# F))$

$\implies \text{clauses } S \models_{pm} C' + \{\#L\# \}$

$\implies F \models_{as} C \text{Not } C'$

$\implies T \sim \text{prepend-trail } (\text{Propagated } L ()) (\text{reduce-trail-to}_{NOT} F S)$

$\implies P S T$

**shows**  $P S T$

**apply** ( $\text{induct } T \text{ rule: } \text{dpll-bj-induct}[\text{OF } \text{local.dpll-with-backjumping-ops-axioms}]$ )

**apply** ( $\text{rule } \text{assms}(1)$ )

**using**  $\text{assms}(3)$  **apply**  $\text{blast}$

**apply** ( $\text{elim } \text{propagateE}$ ) **using**  $\text{assms}(4)$  **apply**  $\text{blast}$

**apply** ( $\text{elim } \text{backjumpE}$ ) **using**  $\text{assms}(5)$   $\langle \text{inv } S \rangle$  **by**  $\text{simp}$

### 14.3.2 Basic properties

**First, some better suited induction principle** **lemma**  $\text{dpll-bj-clauses} :$

**assumes**  $\text{dpll-bj } S T$  **and**  $\text{inv } S$

**shows** *clauses S = clauses T*  
**using** *assms by (induction rule: dpll-bj-all-induct) auto*

**No duplicates in the trail lemma** *dpll-bj-no-dup:*

**assumes** *dpll-bj S T and inv S*  
**and** *no-dup (trail S)*  
**shows** *no-dup (trail T)*  
**using** *assms by (induction rule: dpll-bj-all-induct)*  
*(auto simp add: defined-lit-map reduce-trail-to<sub>NOT</sub>-skip-beginning)*

**Valuations lemma** *dpll-bj-sat-iff:*

**assumes** *dpll-bj S T and inv S*  
**shows** *I  $\models_{sm}$  clauses S  $\longleftrightarrow$  I  $\models_{sm}$  clauses T*  
**using** *assms by (induction rule: dpll-bj-all-induct) auto*

**Clauses lemma** *dpll-bj-atms-of-m-clauses-inv:*

**assumes**  
*dpll-bj S T and*  
*inv S*  
**shows** *atms-of-mu (clauses S) = atms-of-mu (clauses T)*  
**using** *assms by (induction rule: dpll-bj-all-induct) auto*

**lemma** *dpll-bj-atms-in-trail:*

**assumes**  
*dpll-bj S T and*  
*inv S and*  
*atm-of ' (lits-of (trail S))  $\subseteq$  atms-of-mu (clauses S)*  
**shows** *atm-of ' (lits-of (trail T))  $\subseteq$  atms-of-mu (clauses S)*  
**using** *assms by (induction rule: dpll-bj-all-induct)*  
*(auto simp: in-plus-implies-atm-of-on-atms-of-m reduce-trail-to<sub>NOT</sub>-skip-beginning)*

**lemma** *dpll-bj-atms-in-trail-in-set:*

**assumes** *dpll-bj S T and*  
*inv S and*  
*atms-of-mu (clauses S)  $\subseteq$  A and*  
*atm-of ' (lits-of (trail S))  $\subseteq$  A*  
**shows** *atm-of ' (lits-of (trail T))  $\subseteq$  A*  
**using** *assms by (induction rule: dpll-bj-all-induct)*  
*(auto simp: in-plus-implies-atm-of-on-atms-of-m)*

**lemma** *dpll-bj-all-decomposition-implies-inv:*

**assumes**  
*dpll-bj S T and*  
*inv: inv S and*  
*decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*  
**shows** *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*  
**using** *assms(1,2)*

**proof** *(induction rule: dpll-bj-all-induct)*

**case** *decide<sub>NOT</sub>*

**then show** *?case using decomp by auto*

**next**

**case** *(propagate<sub>NOT</sub> C L T) note propa = this(1) and undef = this(3) and T = this(4)*  
**let** *?M' = trail (prepend-trail (Propagated L ())) S)*  
**let** *?N = clauses S*  
**obtain** *a y l where ay: get-all-marked-decomposition ?M' = (a, y) # l*



by (cases get-all-marked-decomposition ?M') fastforce+  
 then have M': ?M' = y @ a using get-all-marked-decomposition-decomp[of ?M'] by auto  
 have M: get-all-marked-decomposition (trail S) = (a, tl y) # l  
 using ay undef by (cases get-all-marked-decomposition (trail S)) auto  
 have y0: y = (Propagated L ()) # (tl y)  
 using ay undef by (auto simp add: M)  
 from arg-cong[OF this, of set] have y[simp]: set y = insert (Propagated L ()) (set (tl y))  
 by simp  
 have tr-S: trail S = tl y @ a  
 using arg-cong[OF M', of tl] y0 M get-all-marked-decomposition-decomp by force  
 have a-Un-N-M: (λa. {#lit-of a#}) ' set a ∪ set-mset ?N ⊨ps (λa. {#lit-of a#}) ' set (tl y)  
 using decomp ay unfolding all-decomposition-implies-def by (simp add: M)+

moreover have (λa. {#lit-of a#}) ' set a ∪ set-mset ?N ⊨p {#L#} (is ?I ⊨p -)  
 proof (rule true-clss-clss-plus-CNot)  
 show ?I ⊨p C + {#L#}  
 using propa propagate<sub>NOT</sub>.prems by (auto dest!: true-clss-clss-in-imp-true-clss-clss)  
 next

have (λm. {#lit-of m#}) ' set ?M' ⊨ps CNot C  
 using (trail S ⊨as CNot C) undef by (auto simp add: true-annots-true-clss-clss)  
 have a1: (λm. {#lit-of m#}) ' set a ∪ (λm. {#lit-of m#}) ' set (tl y) ⊨ps CNot C  
 using propagate<sub>NOT</sub>.hyps(2) tr-S true-annots-true-clss-clss  
 by (force simp add: image-Un sup-commute)  
 have a2: set-mset (clauses S) ∪ (λa. {#lit-of a#}) ' set a  
 ⊨ps (λa. {#lit-of a#}) ' set (tl y)  
 using calculation by (auto simp add: sup-commute)  
 show (λm. {#lit-of m#}) ' set a ∪ set-mset (clauses S) ⊨ps CNot C  
 proof -  
 have set-mset (clauses S) ∪ (λm. {#lit-of m#}) ' set a ⊨ps  
 (λm. {#lit-of m#}) ' set a ∪ (λm. {#lit-of m#}) ' set (tl y)  
 using a2 true-clss-clss-def by blast  
 then show (λm. {#lit-of m#}) ' set a ∪ set-mset (clauses S) ⊨ps CNot C  
 using a1 unfolding sup-commute by (meson true-clss-clss-left-right  
 true-clss-clss-union-and true-clss-clss-union-l-r )

qed  
 qed

ultimately have (λa. {#lit-of a#}) ' set a ∪ set-mset ?N ⊨ps (λa. {#lit-of a#}) ' set ?M'  
 unfolding M' by (auto simp add: all-in-true-clss-clss image-Un)

then show ?case  
 using decomp T M undef unfolding ay all-decomposition-implies-def by (auto simp add: ay)  
 next  
 case (backjump C F' K F L D T) note confl = this(2) and tr = this(3) and undef = this(4)  
 and L = this(5) and N-C = this(6) and vars-D = this(5) and T = this(8)  
 have decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition F)  
 using decomp unfolding tr all-decomposition-implies-def  
 by (metis (no-types, lifting) get-all-marked-decomposition.simps(1)  
 get-all-marked-decomposition-never-empty hd-Cons-tl insert-iff list.sel(3) list.set(2)  
 tl-get-all-marked-decomposition-skip-some)

moreover have (λa. {#lit-of a#}) ' set (fst (hd (get-all-marked-decomposition F)))  
 ∪ set-mset (clauses S)  
 ⊨ps (λa. {#lit-of a#}) ' set (snd (hd (get-all-marked-decomposition F)))  
 by (metis all-decomposition-implies-cons-single decomp get-all-marked-decomposition-never-empty)

```

    hd-Cons-tl)
  moreover
    have vars-of-D: atms-of D  $\subseteq$  atm-of ' lits-of F
    using  $\langle F \models_{as} CNot\ D \rangle$  unfolding atms-of-def
    by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)

  obtain a b li where F: get-all-marked-decomposition F = (a, b) # li
  by (cases get-all-marked-decomposition F) auto
  have F = b @ a
  using get-all-marked-decomposition-decomp[of F a b] F by auto
  have a-N-b:  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set a  $\cup$  set-mset (clauses S)  $\models_{ps}$   $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set b
  using decomp unfolding all-decomposition-implies-def by (auto simp add: F)

  have F-D:  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set F  $\models_{ps}$  CNot D
  using  $\langle F \models_{as} CNot\ D \rangle$  by (simp add: true-annots-true-clss-clss)
  then have  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set a  $\cup$   $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set b  $\models_{ps}$  CNot D
  unfolding  $\langle F = b @ a \rangle$  by (simp add: image-Un sup.commute)
  have a-N-CNot-D:  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set a  $\cup$  set-mset (clauses S)
   $\models_{ps}$  CNot D  $\cup$   $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set b
  apply (rule true-clss-clss-left-right)
  using a-N-b F-D unfolding  $\langle F = b @ a \rangle$  by (auto simp add: image-Un ac-simps)

  have a-N-D-L:  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set a  $\cup$  set-mset (clauses S)  $\models_p$  D+ $\{\#L\# \}$ 
  by (simp add: N-C)
  have  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set a  $\cup$  set-mset (clauses S)  $\models_p$   $\{\#L\# \}$ 
  using a-N-D-L a-N-CNot-D by (blast intro: true-clss-clss-plus-CNot)
  then show ?case
  using decomp T tr undef unfolding all-decomposition-implies-def by (auto simp add: F)
qed

```

### 14.3.3 Termination

**Using a proper measure** lemma length-get-all-marked-decomposition-append-Marked:

```

length (get-all-marked-decomposition (F' @ Marked K () # F)) =
  length (get-all-marked-decomposition F')
+ length (get-all-marked-decomposition (Marked K () # F))
- 1
by (induction F' rule: marked-lit-list-induct) auto

```

**lemma** take-length-get-all-marked-decomposition-marked-sandwich:

```

take (length (get-all-marked-decomposition F))
  (map (f o snd) (rev (get-all-marked-decomposition (F' @ Marked K () # F))))
=
  map (f o snd) (rev (get-all-marked-decomposition F))

```

**proof** (induction F' rule: marked-lit-list-induct)

```

  case nil
  then show ?case by auto
next
  case (marked K)
  then show ?case by (simp add: length-get-all-marked-decomposition-append-Marked)
next
  case (proped L m F') note IH = this(1)
  obtain a b l where F': get-all-marked-decomposition (F' @ Marked K () # F) = (a, b) # l
  by (cases get-all-marked-decomposition (F' @ Marked K () # F)) auto
  have length (get-all-marked-decomposition F) - length l = 0

```

```

    using length-get-all-marked-decomposition-append-Marked[of F' K F]
    unfolding F' by (cases get-all-marked-decomposition F') auto
  then show ?case
    using IH by (simp add: F')
qed

```

**lemma** *length-get-all-marked-decomposition-length*:  
 $\text{length } (\text{get-all-marked-decomposition } M) \leq 1 + \text{length } M$   
 by (induction *M* rule: marked-lit-list-induct) auto

**lemma** *length-in-get-all-marked-decomposition-bounded*:

assumes  $i : i \in \text{set } (\text{trail-weight } S)$   
 shows  $i \leq \text{Suc } (\text{length } (\text{trail } S))$

**proof** –

obtain *a b* where

$(a, b) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$  and

*ib*:  $i = \text{Suc } (\text{length } b)$

using *i* by auto

then obtain *c* where  $\text{trail } S = c @ b @ a$

using *get-all-marked-decomposition-exists-prepend'* by metis

from *arg-cong*[*OF this, of length*] show ?thesis using *i ib* by auto

qed

**Well-foundedness** The bounds are the following:

- $1 + \text{card } (\text{atms-of-}m \ A)$ :  $\text{card } (\text{atms-of-}m \ A)$  is an upper bound on the length of the list. As *get-all-marked-decomposition* appends an possibly empty couple at the end, adding one is needed.
- $2 + \text{card } (\text{atms-of-}m \ A)$ :  $\text{card } (\text{atms-of-}m \ A)$  is an upper bound on the number of elements, where adding one is necessary for the same reason as for the bound on the list, and one is needed to have a strict bound.

**abbreviation** *unassigned-lit* :: '*b* literal multiset set  $\Rightarrow$  '*a* list  $\Rightarrow$  nat **where**

$\text{unassigned-lit } N \ M \equiv \text{card } (\text{atms-of-}m \ N) - \text{length } M$

**lemma** *dpll-bj-trail-mes-increasing-prop*:

fixes *M* :: ('*v*, unit, unit) marked-lits and *N* :: '*v* clauses

assumes

*dpll-bj* *S T* and

*inv* *S* and

*NA*:  $\text{atms-of-}mu \ (\text{clauses } S) \subseteq \text{atms-of-}m \ A$  and

*MA*:  $\text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-}m \ A$  and

*n-d*: *no-dup* (*trail* *S*) and

*finite*: *finite* *A*

shows  $\mu_C \ (1 + \text{card } (\text{atms-of-}m \ A)) \ (2 + \text{card } (\text{atms-of-}m \ A)) \ (\text{trail-weight } T)$

$> \mu_C \ (1 + \text{card } (\text{atms-of-}m \ A)) \ (2 + \text{card } (\text{atms-of-}m \ A)) \ (\text{trail-weight } S)$

using *assms*(1,2)

**proof** (*induction rule*: *dpll-bj-all-induct*)

case (*propagate*<sub>NOT</sub> *C L*) note *CLN* = *this*(1) and *MC* = *this*(2) and *undef-L* = *this*(3) and *T* =

*this*(4)

have *incl*:  $\text{atm-of } ' \text{ lits-of } (\text{Propagated } L \ ()) \ \# \ \text{trail } S \subseteq \text{atms-of-}m \ A$

using *propagate*<sub>NOT</sub>.*hyps* *propagate-ops*.*propagate*<sub>NOT</sub> *dpll-bj-atms-in-trail-in-set* *bj-propagate*<sub>NOT</sub>

*NA* *MA* *CLN* by (auto simp: *in-plus-implies-atm-of-on-atms-of-}m*)

```

have no-dup: no-dup (Propagated L () # trail S)
  using defined-lit-map n-d undef-L by auto
obtain a b l where M: get-all-marked-decomposition (trail S) = (a, b) # l
  by (case-tac get-all-marked-decomposition (trail S)) auto
have b-le-M: length b ≤ length (trail S)
  using get-all-marked-decomposition-decomp[of trail S] by (simp add: M)
have finite (atms-of-m A) using finite by simp

then have length (Propagated L () # trail S) ≤ card (atms-of-m A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of[OF no-dup]
  by (simp add: card-mono)
then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L d # b))
  using b-le-M by auto
then show ?case using T undef-L by (auto simp: latm M  $\mu_C$ -cons)
next
case (decideNOT L) note undef-L = this(1) and MC = this(2) and T = this(3)
have incl: atm-of ' lits-of (Marked L () # (trail S)) ⊆ atms-of-m A
  using dp11-bj-atms-in-trail-in-set bj-decideNOT decideNOT.decideNOT[OF decideNOT.hyps] NA MA
MC
  by auto

have no-dup: no-dup (Marked L () # (trail S))
  using defined-lit-map n-d undef-L by auto
obtain a b l where M: get-all-marked-decomposition (trail S) = (a, b) # l
  by (case-tac get-all-marked-decomposition (trail S)) auto

then have length (Marked L () # (trail S)) ≤ card (atms-of-m A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of[OF no-dup]
  by (simp add: card-mono)
then have latm: unassigned-lit A (trail S) = Suc (unassigned-lit A (Marked L lv # (trail S)))
  by force
show ?case using T undef-L by (simp add: latm  $\mu_C$ -cons)
next
case (backjump C F' K F L C' T) note undef-L = this(4) and MC = this(1) and tr-S = this(3)
and
  L = this(5) and T = this(8)
have incl: atm-of ' lits-of (Propagated L () # F) ⊆ atms-of-m A
  using dp11-bj-atms-in-trail-in-set NA MA tr-S L by auto

have no-dup: no-dup (Propagated L () # F)
  using defined-lit-map n-d undef-L tr-S by auto
obtain a b l where M: get-all-marked-decomposition (trail S) = (a, b) # l
  by (cases get-all-marked-decomposition (trail S)) auto
have b-le-M: length b ≤ length (trail S)
  using get-all-marked-decomposition-decomp[of trail S] by (simp add: M)
have fin-atms-A: finite (atms-of-m A) using finite by simp

then have F-le-A: length (Propagated L () # F) ≤ card (atms-of-m A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of[OF no-dup]
  by (simp add: card-mono)
have tr-S-le-A: length (trail S) ≤ (card (atms-of-m A))
  using n-d MA by (metis fin-atms-A card-mono no-dup-length-eq-card-atm-of-lits-of)
obtain a b l where F: get-all-marked-decomposition F = (a, b) # l
  by (cases get-all-marked-decomposition F) auto
then have F = b @ a

```

```

using get-all-marked-decomposition-decomp[of Propagated L () # F a
  Propagated L () # b] by simp
then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L () # b))
  using F-le-A by simp
obtain rem where
  rem: map (λa. Suc (length (snd a))) (rev (get-all-marked-decomposition (F' @ Marked K () # F)))
  = map (λa. Suc (length (snd a))) (rev (get-all-marked-decomposition F)) @ rem
  using take-length-get-all-marked-decomposition-marked-sandwich[of F λa. Suc (length a) F' K]
  unfolding o-def by (metis append-take-drop-id)
then have rem: map (λa. Suc (length (snd a)))
  (get-all-marked-decomposition (F' @ Marked K () # F))
  = rev rem @ map (λa. Suc (length (snd a))) ((get-all-marked-decomposition F))
  by (simp add: rev-map[symmetric] rev-swap)
have length (rev rem @ map (λa. Suc (length (snd a))) (get-all-marked-decomposition F))
  ≤ Suc (card (atms-of-m A))
  using arg-cong[OF rem, of length] tr-S-le-A
  length-get-all-marked-decomposition-length[of F' @ Marked K () # F] tr-S by auto
moreover
  { fix i :: nat and xs :: 'a list
    have i < length xs ⇒ length xs - Suc i < length xs
      by auto
    then have H: i < length xs ⇒ rev xs ! i ∈ set xs
      using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)
  } note H = this
have ∀ i < length rem. rev rem ! i < card (atms-of-m A) + 2
  using tr-S-le-A length-in-get-all-marked-decomposition-bounded[of - S] unfolding tr-S
  by (force simp add: o-def rem dest!: H intro: length-get-all-marked-decomposition-length)
ultimately show ?case
  using μC-bounded[of rev rem card (atms-of-m A)+2 unassigned-lit A l] T undef-L
  by (simp add: rem μC-append μC-cons F tr-S)
qed

```

**lemma** dpll-bj-trail-mes-decreasing-prop:

**assumes** dpll: dpll-bj S T **and** inv: inv S **and**  
 N-A: atms-of-mu (clauses S) ⊆ atms-of-m A **and**  
 M-A: atm-of ' lits-of (trail S) ⊆ atms-of-m A **and**  
 nd: no-dup (trail S) **and**  
 fin-A: finite A

**shows** (2+card (atms-of-m A)) ^ (1+card (atms-of-m A))  
 - μ<sub>C</sub> (1+card (atms-of-m A)) (2+card (atms-of-m A)) (trail-weight T)  
 < (2+card (atms-of-m A)) ^ (1+card (atms-of-m A))  
 - μ<sub>C</sub> (1+card (atms-of-m A)) (2+card (atms-of-m A)) (trail-weight S)

**proof** -

```

let ?b = 2+card (atms-of-m A)
let ?s = 1+card (atms-of-m A)
let ?μ = μC ?s ?b
have M'-A: atm-of ' lits-of (trail T) ⊆ atms-of-m A
  by (meson M-A N-A dpll dpll-bj-atms-in-trail-in-set inv)
have nd': no-dup (trail T)
  using ⟨dpll-bj S T⟩ dpll-bj-no-dup nd inv by blast
{ fix i :: nat and xs :: 'a list
  have i < length xs ⇒ length xs - Suc i < length xs
    by auto
  then have H: i < length xs ⇒ xs ! i ∈ set xs
    using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)

```

```

} note H = this

have l-M-A: length (trail S) ≤ card (atms-of-m A)
  by (simp add: fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of nd)
have l-M'-A: length (trail T) ≤ card (atms-of-m A)
  by (simp add: fin-A M'-A card-mono no-dup-length-eq-card-atm-of-lits-of nd')
have l-trail-weight-M: length (trail-weight T) ≤ 1 + card (atms-of-m A)
  using l-M'-A length-get-all-marked-decomposition-length[of trail T] by auto
have bounded-M: ∀ i < length (trail-weight T). (trail-weight T)! i < card (atms-of-m A) + 2
  using length-in-get-all-marked-decomposition-bounded[of - T] l-M'-A
  by (metis (no-types, lifting) Nat.le-trans One-nat-def Suc-1 add.right-neutral add-Suc-right
    le-imp-less-Suc less-eq-Suc-le nth-mem)

from dp11-bj-trail-mes-increasing-prop[OF dp11 inv N-A M-A nd fin-A]
have μC ?s ?b (trail-weight S) < μC ?s ?b (trail-weight T) by simp
moreover from μC-bounded[OF bounded-M l-trail-weight-M]
  have μC ?s ?b (trail-weight T) ≤ ?b ^ ?s by auto
ultimately show ?thesis by linarith
qed

lemma wf-dp11-bj:
  assumes fin: finite A
  shows wf {(T, S). dp11-bj S T
    ∧ atms-of-mu (clauses S) ⊆ atms-of-m A ∧ atm-of ' lits-of (trail S) ⊆ atms-of-m A
    ∧ no-dup (trail S) ∧ inv S}
  (is wf ?A)
proof (rule wf-bounded-measure[of -
  λ-. (2 + card (atms-of-m A)) ^ (1 + card (atms-of-m A))
  λS. μC (1 + card (atms-of-m A)) (2 + card (atms-of-m A)) (trail-weight S)])
  fix a b :: 'st
  let ?b = 2 + card (atms-of-m A)
  let ?s = 1 + card (atms-of-m A)
  let ?μ = μC ?s ?b
  assume ab: (b, a) ∈ {(T, S). dp11-bj S T
    ∧ atms-of-mu (clauses S) ⊆ atms-of-m A ∧ atm-of ' lits-of (trail S) ⊆ atms-of-m A
    ∧ no-dup (trail S) ∧ inv S}
  have fin-A: finite (atms-of-m A)
    using fin by auto
  have
    dp11-bj: dp11-bj a b and
    N-A: atms-of-mu (clauses a) ⊆ atms-of-m A and
    M-A: atm-of ' lits-of (trail a) ⊆ atms-of-m A and
    nd: no-dup (trail a) and
    inv: inv a
    using ab by auto

  have M'-A: atm-of ' lits-of (trail b) ⊆ atms-of-m A
    by (meson M-A N-A ⟨dp11-bj a b⟩ dp11-bj-atms-in-trail-in-set inv)
  have nd': no-dup (trail b)
    using ⟨dp11-bj a b⟩ dp11-bj-no-dup nd inv by blast
  { fix i :: nat and xs :: 'a list
    have i < length xs ⇒ length xs - Suc i < length xs
      by auto
    then have H: i < length xs ⇒ xs ! i ∈ set xs

```

```

    using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)
  } note H = this

have l-M-A: length (trail a) ≤ card (atms-of-m A)
  by (simp add: fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of nd)
have l-M'-A: length (trail b) ≤ card (atms-of-m A)
  by (simp add: fin-A M'-A card-mono no-dup-length-eq-card-atm-of-lits-of nd')
have l-trail-weight-M: length (trail-weight b) ≤ 1 + card (atms-of-m A)
  using l-M'-A length-get-all-marked-decomposition-length[of trail b] by auto
have bounded-M: ∀ i < length (trail-weight b). (trail-weight b)! i < card (atms-of-m A) + 2
  using length-in-get-all-marked-decomposition-bounded[of - b] l-M'-A
  by (metis (no-types, lifting) Nat.le-trans One-nat-def Suc-1 add.right-neutral add-Suc-right
    le-imp-less-Suc less-eq-Suc-le nth-mem)

from dpll-bj-trail-mes-increasing-prop[OF dpll-bj inv N-A M-A nd fin]
have μC ?s ?b (trail-weight a) < μC ?s ?b (trail-weight b) by simp
moreover from μC-bounded[OF bounded-M l-trail-weight-M]
  have μC ?s ?b (trail-weight b) ≤ ?b ^ ?s by auto
ultimately show ?b ^ ?s ≤ ?b ^ ?s ∧
  μC ?s ?b (trail-weight b) ≤ ?b ^ ?s ∧
  μC ?s ?b (trail-weight a) < μC ?s ?b (trail-weight b)
  by blast
qed

```

#### 14.3.4 Normal Forms

We prove that given a normal form of DPLL, with some invariants, the 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* rules tells us that every variable in  $N$  has a value.
2.  $\neg M \models_{as} N$  tells us that there is conflict.
3. There is at least one decision in the trail (otherwise,  $M$  is a model of  $N$ ).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

The assumption are saying that we have a finite upper bound  $A$  for the literals, that we cannot do any step *no-step dpll-bj*  $S$

**theorem** *dpll-backjump-final-state*:

**fixes**  $A :: 'v \text{ literal multiset set}$  **and**  $S \ T :: 'st$

**assumes**

$atms-of-\mu (clauses \ S) \subseteq atms-of-m \ A$  **and**

$atm-of \text{ ' } lits-of (trail \ S) \subseteq atms-of-m \ A$  **and**

$no-dup (trail \ S)$  **and**

$finite \ A$  **and**

$inv: inv \ S$  **and**

$n-s: no-step \ dpll-bj \ S$  **and**

$decomp: all-decomposition-implies-m (clauses \ S) (get-all-marked-decomposition (trail \ S))$

**shows**  $unsatisfiable (set-mset (clauses \ S))$

$\vee (trail \ S \models_{asm} clauses \ S \wedge satisfiable (set-mset (clauses \ S)))$

```

proof –
  let ?N = set-mset (clauses S)
  let ?M = trail S
  consider
    (sat) satisfiable ?N and ?M  $\models_{as}$  ?N
    | (sat') satisfiable ?N and  $\neg$  ?M  $\models_{as}$  ?N
    | (unsat) unsatisfiable ?N
  by auto
  then show ?thesis
  proof cases
    case sat' note sat = this(1) and M = this(2)
    obtain C where C  $\in$  ?N and  $\neg$  ?M  $\models_a$  C using M unfolding true-annots-def by auto
    obtain I :: 'v literal set where
      I  $\models_s$  ?N and
      cons: consistent-interp I and
      tot: total-over-m I ?N and
      atm-I-N: atm-of 'I  $\subseteq$  atms-of-m ?N
      using sat unfolding satisfiable-def-min by auto
    let ?I = I  $\cup$  {P | P. P  $\in$  lits-of ?M  $\wedge$  atm-of P  $\notin$  atm-of 'I}
    let ?O = {{#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set ?M  $\wedge$  atm-of (lit-of L)  $\notin$  atms-of-m ?N}
    have cons-I': consistent-interp ?I
      using cons using (no-dup ?M) unfolding consistent-interp-def
      by (auto simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def
        dest!: no-dup-cannot-not-lit-and-uminus)
    have tot-I': total-over-m ?I (?N  $\cup$  ( $\lambda$ a. {#lit-of a#}) ' set ?M)
      using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
      by fastforce
    have {P | P. P  $\in$  lits-of ?M  $\wedge$  atm-of P  $\notin$  atm-of 'I}  $\models_s$  ?O
      using (I  $\models_s$  ?N) atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)
    then have I'-N: ?I  $\models_s$  ?N  $\cup$  ?O
      using (I  $\models_s$  ?N) true-clss-union-increase by force
    have tot': total-over-m ?I (?N  $\cup$  ?O)
      using atm-I-N tot unfolding total-over-m-def total-over-set-def
      by (force simp: image-iff lits-of-def dest!: is-marked-ex-Marked)

    have atms-N-M: atms-of-m ?N  $\subseteq$  atm-of ' lits-of ?M
    proof (rule ccontr)
      assume  $\neg$  ?thesis
      then obtain l :: 'v where
        l-N: l  $\in$  atms-of-m ?N and
        l-M: l  $\notin$  atm-of ' lits-of ?M
      by auto
      have undefined-lit ?M (Pos l)
        using l-M by (metis Marked-Propagated-in-iff-in-lits-of
          atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
      from bj-decideNOT[OF decideNOT[OF this]] show False
      using l-N n-s by (metis literal.sel(1) state-eqNOT-ref)
    qed

    have ?M  $\models_{as}$  CNot C
      by (metis atms-N-M (C  $\in$  ?N) ( $\neg$  ?M  $\models_a$  C) all-variables-defined-not-imply-cnot
        atms-of-atms-of-m-mono atms-of-m-CNot-atms-of atms-of-m-CNot-atms-of-m subsetCE)
    have  $\exists$  l  $\in$  set ?M. is-marked l
    proof (rule ccontr)
      let ?O = {{#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set ?M  $\wedge$  atm-of (lit-of L)  $\notin$  atms-of-m ?N}

```



```

have  $\vartheta[\text{iff}]$ :  $\bigwedge I. \text{total-over-m } I \ (\vartheta N \cup \vartheta O \cup (\lambda a. \{\# \text{lit-of } a\# \})) \text{ ' set } \vartheta M$ 
 $\longleftrightarrow \text{total-over-m } I \ (\vartheta N \cup (\lambda a. \{\# \text{lit-of } a\# \})) \text{ ' set } \vartheta M$ 
unfolding total-over-set-def total-over-m-def atms-of-m-def by auto
assume  $\neg \vartheta \text{thesis}$ 
then have  $[\text{simp}]: \{\{\# \text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } \vartheta M\}$ 
 $= \{\{\# \text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } \vartheta M \wedge \text{atm-of } (\text{lit-of } L) \notin \text{atms-of-m } \vartheta N\}$ 
by auto
then have  $\vartheta N \cup \vartheta O \models_{ps} (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } \vartheta M$ 
using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

then have  $\vartheta I \models_s (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } \vartheta M$ 
using cons-I' I'-N tot-I'  $\langle \vartheta I \models_s \vartheta N \cup \vartheta O \rangle$  unfolding  $\vartheta$  true-clss-clss-def by blast
then have  $\text{lits-of } \vartheta M \subseteq \vartheta I$ 
unfolding true-clss-def lits-of-def by auto
then have  $\vartheta M \models_{as} \vartheta N$ 
using  $I'-N \langle C \in \vartheta N \rangle \langle \neg \vartheta M \models_a C \rangle$  cons-I' atms-N-M
by (meson  $\langle \text{trail } S \models_{as} C \text{Not } C \rangle$  consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
true-annots-def true-clss-mono-set-mset-l true-clss-def)
then show False using M by fast
qed
from List.split-list-first-propE[OF this] obtain  $K :: 'v \text{ literal and}$ 
 $F F' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list where}$ 
 $M\text{-}K: \vartheta M = F' @ \text{Marked } K \ () \# F$  and
 $nm: \forall f \in \text{set } F'. \neg \text{is-marked } f$ 
unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let  $\vartheta K = \text{Marked } K \ () :: ('v, \text{unit}, \text{unit}) \text{ marked-lit}$ 
have  $\vartheta K \in \text{set } \vartheta M$ 
unfolding M-K by auto
let  $\vartheta C = \text{image-mset lit-of } \{\# L \in \# \text{mset } \vartheta M. \text{is-marked } L \wedge L \neq \vartheta K\# \} :: 'v \text{ literal multiset}$ 
let  $\vartheta C' = \text{set-mset } (\text{image-mset } (\lambda L :: 'v \text{ literal. } \{\# L\# \}) (\vartheta C + \{\# \text{lit-of } \vartheta K\# \}))$ 
have  $\vartheta N \cup \{\{\# \text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } \vartheta M\} \models_{ps} (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } \vartheta M$ 
using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have  $C': \vartheta C' = \{\{\# \text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } \vartheta M\}$ 
unfolding M-K apply standard
apply force
using IntI by auto
ultimately have  $N\text{-}C\text{-}M: \vartheta N \cup \vartheta C' \models_{ps} (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } \vartheta M$ 
by auto
have  $N\text{-}M\text{-}False: \vartheta N \cup (\lambda L. \{\# \text{lit-of } L\# \}) \text{ ' (set } \vartheta M) \models_{ps} \{\{\#\}\}$ 
using  $M \langle \vartheta M \models_{as} C \text{Not } C \rangle \langle C \in \vartheta N \rangle$  unfolding true-clss-clss-def true-annots-def Ball-def
true-annot-def by (metis consistent-CNot-not sup.orderE sup-commute true-clss-def
true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using  $\langle \text{no-dup } \vartheta M \rangle$  unfolding M-K by (simp add: defined-lit-map)
moreover
have  $\vartheta N \cup \vartheta C' \models_{ps} \{\{\#\}\}$ 
proof –
have  $A: \vartheta N \cup \vartheta C' \cup (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } \vartheta M =$ 
 $\vartheta N \cup (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } \vartheta M$ 
unfolding M-K by auto
show ?thesis
using true-clss-clss-left-right[OF N-C-M, of  $\{\{\#\}\}$ ] N-M-False unfolding A by auto
qed
have  $\vartheta N \models_p \text{image-mset uminus } \vartheta C + \{\# - K\# \}$ 
unfolding true-clss-clss-def true-clss-clss-def total-over-m-def

```

```

proof (intro allI impI)
  fix I
  assume
    tot: total-over-set I (atms-of-m (?N  $\cup$  {image-mset uminus ?C+ {#- K#}})) and
    cons: consistent-interp I and
    I  $\models_s$  ?N
  have (K  $\in$  I  $\wedge$  -K  $\notin$  I)  $\vee$  (-K  $\in$  I  $\wedge$  K  $\notin$  I)
    using cons tot unfolding consistent-interp-def by (cases K) auto
  have tot': total-over-set I
    (atm-of 'lit-of ' (set ?M  $\cap$  {L. is-marked L  $\wedge$  L  $\neq$  Marked K ()}))
    using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
  { fix x :: ('v, unit, unit) marked-lit
    assume
      a3: lit-of x  $\notin$  I and
      a1: x  $\in$  set ?M and
      a4: is-marked x and
      a5: x  $\neq$  Marked K ()
    then have Pos (atm-of (lit-of x))  $\in$  I  $\vee$  Neg (atm-of (lit-of x))  $\in$  I
      using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
    moreover have f6: Neg (atm-of (lit-of x)) = - Pos (atm-of (lit-of x))
      by simp
    ultimately have - lit-of x  $\in$  I
      using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
        literal.sel(1))
    } note H = this

  have  $\neg I \models_s ?C'$ 
    using ⟨?N  $\cup$  ?C'  $\models_{ps}$  {{#}}⟩ tot cons ⟨I  $\models_s$  ?N⟩
    unfolding true-clss-clss-def total-over-m-def
    by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-m-single-image-atm-of-lit-of)
  then show I  $\models$  image-mset uminus ?C + {#- K#}
    unfolding true-clss-def true-cls-def Bex-mset-def
    using ⟨(K  $\in$  I  $\wedge$  -K  $\notin$  I)  $\vee$  (-K  $\in$  I  $\wedge$  K  $\notin$  I)⟩
    by (auto dest!: H)
  qed

moreover have F  $\models_{as}$  CNot (image-mset uminus ?C)
  using nm unfolding true-annots-def CNot-def M-K by (auto simp add: lits-of-def)
ultimately have False
  using bj-can-jump[of S F' K F C -K
    image-mset uminus (image-mset lit-of {# L :# mset ?M. is-marked L  $\wedge$  L  $\neq$  Marked K ()#})]
    ⟨C $\in$ ?N⟩ n-s ⟨?M  $\models_{as}$  CNot C⟩ bj-backjump inv ⟨no-dup (trail S)⟩ unfolding M-K by auto
  then show ?thesis by fast
qed auto
qed

end

locale dpll-with-backjumping =
  dpll-with-backjumping-ops trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
  propagate-conds inv backjump-conds
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT remove-clsNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and

```

```

    propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
    inv :: 'st  $\Rightarrow$  bool and
    backjump-conds :: 'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
+
assumes dpll-bj-inv:  $\bigwedge S T. \text{dpll-bj } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 
begin

lemma rtrancpl-dpll-bj-inv:
assumes dpll-bj**  $S T$  and inv  $S$ 
shows inv  $T$ 
using assms by (induction rule: rtrancpl-induct)
    (auto simp add: dpll-bj-no-dup intro: dpll-bj-inv)

lemma rtrancpl-dpll-bj-no-dup:
assumes dpll-bj**  $S T$  and inv  $S$ 
and no-dup (trail  $S$ )
shows no-dup (trail  $T$ )
using assms by (induction rule: rtrancpl-induct)
    (auto simp add: dpll-bj-no-dup dest: rtrancpl-dpll-bj-inv dpll-bj-inv)

lemma rtrancpl-dpll-bj-atms-of-m-clauses-inv:
assumes
    dpll-bj**  $S T$  and inv  $S$ 
shows atms-of-mu (clauses  $S$ ) = atms-of-mu (clauses  $T$ )
using assms by (induction rule: rtrancpl-induct)
    (auto dest: rtrancpl-dpll-bj-inv dpll-bj-atms-of-m-clauses-inv)

lemma rtrancpl-dpll-bj-atms-in-trail:
assumes
    dpll-bj**  $S T$  and
    inv  $S$  and
    atm-of ' (lits-of (trail  $S$ ))  $\subseteq$  atms-of-mu (clauses  $S$ )
shows atm-of ' (lits-of (trail  $T$ ))  $\subseteq$  atms-of-mu (clauses  $T$ )
using assms apply (induction rule: rtrancpl-induct)
using dpll-bj-atms-in-trail dpll-bj-atms-of-m-clauses-inv rtrancpl-dpll-bj-inv by auto

lemma rtrancpl-dpll-bj-sat-iff:
assumes dpll-bj**  $S T$  and inv  $S$ 
shows  $I \models_{sm} \text{clauses } S \longleftrightarrow I \models_{sm} \text{clauses } T$ 
using assms by (induction rule: rtrancpl-induct)
    (auto dest!: dpll-bj-sat-iff simp: rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-atms-in-trail-in-set:
assumes
    dpll-bj**  $S T$  and
    inv  $S$ 
    atms-of-mu (clauses  $S$ )  $\subseteq A$  and
    atm-of ' (lits-of (trail  $S$ ))  $\subseteq A$ 
shows atm-of ' (lits-of (trail  $T$ ))  $\subseteq A$ 
using assms
by (induction rule: rtrancpl-induct)
    (auto dest: rtrancpl-dpll-bj-inv
        simp add: dpll-bj-atms-in-trail-in-set rtrancpl-dpll-bj-atms-of-m-clauses-inv
        rtrancpl-dpll-bj-inv)

```

**lemma** *rtranclp-dpll-bj-all-decomposition-implies-inv*:

**assumes**

*dpll-bj\*\* S T and*

*inv S*

*all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

**shows** *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*

**using** *assms by (induction rule: rtranclp-induct)*

*(auto intro: dpll-bj-all-decomposition-implies-inv simp: rtranclp-dpll-bj-inv)*

**lemma** *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*:

$\{(T, S). \text{dpll-bj}^{++} S T$

$\wedge \text{atms-of-mu (clauses S)} \subseteq \text{atms-of-m } A \wedge \text{atm-of ' lits-of (trail S)} \subseteq \text{atms-of-m } A$

$\wedge \text{no-dup (trail S)} \wedge \text{inv S}\}$

$\subseteq \{(T, S). \text{dpll-bj } S T \wedge \text{atms-of-mu (clauses S)} \subseteq \text{atms-of-m } A$

$\wedge \text{atm-of ' lits-of (trail S)} \subseteq \text{atms-of-m } A \wedge \text{no-dup (trail S)} \wedge \text{inv S}\}^+$

$(\text{is } ?A \subseteq ?B^+)$

**proof** *standard*

**fix** *x*

**assume** *x-A: x ∈ ?A*

**obtain** *S T::'st where*

*x[simp]: x = (T, S) by (cases x) auto*

**have**

*dpll-bj^{++} S T and*

*atms-of-mu (clauses S) ⊆ atms-of-m A and*

*atm-of ' lits-of (trail S) ⊆ atms-of-m A and*

*no-dup (trail S) and*

*inv S*

**using** *x-A by auto*

**then show** *x ∈ ?B^+ unfolding x*

**proof** *(induction rule: tranclp-induct)*

**case** *base*

**then show** *?case by auto*

**next**

**case** *(step T U) note step = this(1) and ST = this(2) and IH = this(3)[OF this(4-7)]*

*and N-A = this(4) and M-A = this(5) and nd = this(6) and inv = this(7)*

**have** *[simp]: atms-of-mu (clauses S) = atms-of-mu (clauses T)*

**using** *step rtranclp-dpll-bj-atms-of-m-clauses-inv tranclp-into-rtranclp inv by fastforce*

**have** *no-dup (trail T)*

**using** *local.step nd rtranclp-dpll-bj-no-dup tranclp-into-rtranclp inv by fastforce*

**moreover have** *atm-of ' (lits-of (trail T)) ⊆ atms-of-m A*

**by** *(metis inv M-A N-A local.step rtranclp-dpll-bj-atms-in-trail-in-set*

*tranclp-into-rtranclp)*

**moreover have** *inv T*

**using** *inv local.step rtranclp-dpll-bj-inv tranclp-into-rtranclp by fastforce*

**ultimately have** *(U, T) ∈ ?B using ST N-A M-A inv by auto*

**then show** *?case using IH by (rule trancl-into-trancl2)*

**qed**

**qed**

**lemma** *wf-tranclp-dpll-bj*:

**assumes** *fin: finite A*

**shows** *wf {(T, S). dpll-bj^{++} S T*

$\wedge \text{atms-of-mu (clauses S)} \subseteq \text{atms-of-m } A \wedge \text{atm-of ' lits-of (trail S)} \subseteq \text{atms-of-m } A$

$\wedge \text{no-dup (trail S)} \wedge \text{inv S}\}$

**using** *wf-trancl*[*OF wf-dpll-bj*[*OF fin*]] *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*  
**by** (*rule wf-subset*)

**lemma** *dpll-bj-sat-ext-iff*:

*dpll-bj S T  $\implies$  inv S  $\implies$  I $\models_{\text{sextm}}$  clauses S  $\longleftrightarrow$  I $\models_{\text{sextm}}$  clauses T*  
**by** (*simp add: dpll-bj-clauses*)

**lemma** *rtranclp-dpll-bj-sat-ext-iff*:

*dpll-bj\*\* S T  $\implies$  inv S  $\implies$  I $\models_{\text{sextm}}$  clauses S  $\longleftrightarrow$  I $\models_{\text{sextm}}$  clauses T*  
**by** (*induction rule: rtranclp-induct*) (*simp-all add: rtranclp-dpll-bj-inv dpll-bj-sat-ext-iff*)

**theorem** *full-dpll-backjump-final-state*:

**fixes** *A* :: '*v* literal multiset set **and** *S T* :: '*st*

**assumes**

*full*: *full dpll-bj S T and*

*atms-S*: *atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and*

*atms-trail*: *atm-of ' lits-of (trail S)  $\subseteq$  atms-of-m A and*

*n-d*: *no-dup (trail S) and*

*finite A and*

*inv*: *inv S and*

*decomp*: *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

**shows** *unsatisfiable (set-mset (clauses S))*

$\vee$  (*trail T  $\models_{\text{asm}}$  clauses S  $\wedge$  satisfiable (set-mset (clauses S))*)

**proof** –

**have** *st*: *dpll-bj\*\* S T and no-step dpll-bj T*

**using** *full unfolding full-def by fast+*

**moreover have** *atms-of-mu (clauses T)  $\subseteq$  atms-of-m A*

**using** *atms-S inv rtranclp-dpll-bj-atms-of-m-clauses-inv st by blast*

**moreover have** *atm-of ' lits-of (trail T)  $\subseteq$  atms-of-m A*

**using** *atms-S atms-trail inv rtranclp-dpll-bj-atms-in-trail-in-set st by auto*

**moreover have** *no-dup (trail T)*

**using** *n-d inv rtranclp-dpll-bj-no-dup st by blast*

**moreover have** *inv: inv T*

**using** *inv rtranclp-dpll-bj-inv st by blast*

**moreover**

**have** *decomp*: *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*

**using** (*inv S*) *decomp rtranclp-dpll-bj-all-decomposition-implies-inv st by blast*

**ultimately have** *unsatisfiable (set-mset (clauses T))*

$\vee$  (*trail T  $\models_{\text{asm}}$  clauses T  $\wedge$  satisfiable (set-mset (clauses T))*)

**using** (*finite A*) *dpll-backjump-final-state by force*

**then show** *?thesis*

**by** (*meson (inv S) rtranclp-dpll-bj-sat-iff satisfiable-carac st true-annots-true-cls*)

**qed**

**corollary** *full-dpll-backjump-final-state-from-init-state*:

**fixes** *A* :: '*v* literal multiset set **and** *S T* :: '*st*

**assumes**

*full*: *full dpll-bj S T and*

*trail S* = [] **and**

*clauses S* = *N and*

*inv S*

**shows** *unsatisfiable (set-mset N)  $\vee$  (trail T  $\models_{\text{asm}}$  N  $\wedge$  satisfiable (set-mset N))*

**using** *assms full-dpll-backjump-final-state[of S T set-mset N] by auto*

**lemma** *tranclp-dpll-bj-trail-mes-decreasing-prop*:

```

assumes dpll: dpll-bj++ S T and inv: inv S and
N-A: atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and
M-A: atm-of ‘ lits-of (trail S)  $\subseteq$  atms-of-m A and
n-d: no-dup (trail S) and
fin-A: finite A
shows ( $2 + \text{card } (\text{atms-of-m } A) \wedge (1 + \text{card } (\text{atms-of-m } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-m } A)) (2 + \text{card } (\text{atms-of-m } A)) (\text{trail-weight } T)$ 
   $< (2 + \text{card } (\text{atms-of-m } A) \wedge (1 + \text{card } (\text{atms-of-m } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-m } A)) (2 + \text{card } (\text{atms-of-m } A)) (\text{trail-weight } S)$ 
using dpll
proof (induction)
case base
then show ?case
  using N-A M-A n-d dpll-bj-trail-mes-decreasing-prop fin-A inv by blast
next
case (step T U) note st = this(1) and dpll = this(2) and IH = this(3)
have atms-of-mu (clauses S) = atms-of-mu (clauses T)
  using rtranclp-dpll-bj-atms-of-m-clauses-inv by (metis dpll-bj-clauses dpll-bj-inv inv st
    tranclpD)
then have N-A': atms-of-mu (clauses T)  $\subseteq$  atms-of-m A
  using N-A by auto
moreover have M-A': atm-of ‘ lits-of (trail T)  $\subseteq$  atms-of-m A
  by (meson M-A N-A inv rtranclp-dpll-bj-atms-in-trail-in-set st dpll
    tranclp.r-into-trancl tranclp-into-rtranclp tranclp-trans)
moreover have nd: no-dup (trail T)
  by (metis inv n-d rtranclp-dpll-bj-no-dup st tranclp-into-rtranclp)
moreover have inv T
  by (meson dpll dpll-bj-inv inv rtranclp-dpll-bj-inv st tranclp-into-rtranclp)
ultimately show ?case
  using IH dpll-bj-trail-mes-decreasing-prop[of T U A] dpll fin-A by linarith
qed

end

```

## 14.4 CDCL

### 14.4.1 Learn and Forget

```

locale learn-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT remove-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  learn-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool

begin
inductive learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  clauses S  $\models_{pm}$  C  $\Longrightarrow$  atms-of C  $\subseteq$  atms-of-mu (clauses S)  $\cup$  atm-of ‘ (lits-of (trail S))
   $\Longrightarrow$  learn-cond C S
   $\Longrightarrow$  T  $\sim$  add-clsNOT C S
   $\Longrightarrow$  learn S T
inductive-cases learnE: learn S T

```

```

lemma learn- $\mu_C$ -stable:
  assumes learn  $S$   $T$  and no-dup (trail  $S$ )
  shows  $\mu_C$   $A$   $B$  (trail-weight  $S$ ) =  $\mu_C$   $A$   $B$  (trail-weight  $T$ )
  using assms by (auto elim: learnE)
end

locale forget-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
for
  trail :: ' $st \Rightarrow (v, unit, unit)$  marked-lits and
  clauses :: ' $st \Rightarrow v$  clauses and
  prepend-trail :: ' $(v, unit, unit)$  marked-lit  $\Rightarrow st \Rightarrow st$  and tl-trail :: ' $st \Rightarrow st$  and
  add-clNOT remove-clNOT :: ' $v$  clause  $\Rightarrow st \Rightarrow st$  +
fixes
  forget-cond :: ' $v$  clause  $\Rightarrow st \Rightarrow bool$ 
begin
inductive forgetNOT :: ' $st \Rightarrow st \Rightarrow bool$  where
forgetNOT:clauses  $S$  - replicate-mset (count (clauses  $S$ )  $C$ )  $C \models_{pm} C$ 
 $\Rightarrow$  forget-cond  $C$   $S$ 
 $\Rightarrow C \in \#$  clauses  $S$ 
 $\Rightarrow T \sim$  remove-clNOT  $C$   $S$ 
 $\Rightarrow$  forgetNOT  $S$   $T$ 
inductive-cases forgetE: forgetNOT  $S$   $T$ 

lemma forget- $\mu_C$ -stable:
  assumes forgetNOT  $S$   $T$ 
  shows  $\mu_C$   $A$   $B$  (trail-weight  $S$ ) =  $\mu_C$   $A$   $B$  (trail-weight  $T$ )
  using assms by (auto elim!: forgetE)
end

locale learn-and-forgetNOT =
  learn-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT learn-cond +
  forget-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT forget-cond
for
  trail :: ' $st \Rightarrow (v, unit, unit)$  marked-lits and
  clauses :: ' $st \Rightarrow v$  clauses and
  prepend-trail :: ' $(v, unit, unit)$  marked-lit  $\Rightarrow st \Rightarrow st$  and
  tl-trail :: ' $st \Rightarrow st$  and
  add-clNOT remove-clNOT :: ' $v$  clause  $\Rightarrow st \Rightarrow st$  and
  learn-cond forget-cond :: ' $v$  clause  $\Rightarrow st \Rightarrow bool$ 
begin
inductive learn-and-forgetNOT :: ' $st \Rightarrow st \Rightarrow bool$ 
where
lf-learn: learn  $S$   $T \Rightarrow$  learn-and-forgetNOT  $S$   $T$  |
lf-forget: forgetNOT  $S$   $T \Rightarrow$  learn-and-forgetNOT  $S$   $T$ 
end

```

#### 14.4.2 Definition of CDCL

```

locale conflict-driven-clause-learning-ops =
  dpll-with-backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv backjump-conds +
  learn-and-forgetNOT trail clauses prepend-trail tl-trail add-clNOT remove-clNOT learn-cond
  forget-cond
for
  trail :: ' $st \Rightarrow (v, unit, unit)$  marked-lits and

```

```

  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  learn-cond forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

inductive cdclNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
c-dpll-bj: dpll-bj S S'  $\Rightarrow$  cdclNOT S S' |
c-learn: learn S S'  $\Rightarrow$  cdclNOT S S' |
c-forgetNOT: forgetNOT S S'  $\Rightarrow$  cdclNOT S S'

lemma cdclNOT-all-induct[consumes 1, case-names dpll-bj learn forgetNOT]:
fixes S T :: 'st
assumes cdclNOT S T and
dpll:  $\bigwedge T. \text{dpll-bj } S \ T \Rightarrow P \ S \ T$  and
learning:
 $\bigwedge C \ T. \text{clauses } S \models_{pm} C \Rightarrow$ 
 $\text{atms-of } C \subseteq \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S)) \Rightarrow$ 
 $T \sim \text{add-cl}_{NOT} \ C \ S \Rightarrow$ 
 $P \ S \ T$  and
forgetting:  $\bigwedge C \ T. \text{clauses } S - \text{replicate-mset } (\text{count } (\text{clauses } S) \ C) \ C \models_{pm} C \Rightarrow$ 
 $C \in \# \text{clauses } S \Rightarrow$ 
 $T \sim \text{remove-cl}_{NOT} \ C \ S \Rightarrow$ 
 $P \ S \ T$ 
shows P S T
using assms(1) by (induction rule: cdclNOT.induct)
(auto intro: assms(2, 3, 4) elim!: learnE forgetE)+

lemma cdclNOT-no-dup:
assumes
cdclNOT S T and
inv S and
no-dup (trail S)
shows no-dup (trail T)
using assms by (induction rule: cdclNOT-all-induct) (auto intro: dpll-bj-no-dup)

Consistency of the trail lemma cdclNOT-consistent:
assumes
cdclNOT S T and
inv S and
no-dup (trail S)
shows consistent-interp (lits-of (trail T))
using cdclNOT-no-dup[OF assms] distinctconsistent-interp by fast

```

The subtle problem here is that tautologies can be removed, meaning that some variable can disappear of the problem. It is also possible that some variable of the trail are not in the clauses anymore.

```

lemma cdclNOT-atms-of-m-clauses-decreasing:
assumes cdclNOT S T and inv S and no-dup (trail S)
shows atms-of-mu (clauses T)  $\subseteq$  atms-of-mu (clauses S)  $\cup$  atm-of ' (lits-of (trail S))
using assms by (induction rule: cdclNOT-all-induct)

```



(*auto dest!*: *dpll-bj-atms-of-m-clauses-inv set-mp simp add: atms-of-m-def Union-eq*)

**lemma** *cdcl<sub>NOT</sub>-atms-in-trail*:

**assumes** *cdcl<sub>NOT</sub> S T and inv S and no-dup (trail S)*  
**and** *atm-of ‘ (lits-of (trail S))  $\subseteq$  atms-of-mu (clauses S)*  
**shows** *atm-of ‘ (lits-of (trail T))  $\subseteq$  atms-of-mu (clauses S)*  
**using** *assms by (induction rule: cdcl<sub>NOT</sub>-all-induct) (auto simp add: dpll-bj-atms-in-trail)*

**lemma** *cdcl<sub>NOT</sub>-atms-in-trail-in-set*:

**assumes**  
*cdcl<sub>NOT</sub> S T and inv S and no-dup (trail S) and*  
*atms-of-mu (clauses S)  $\subseteq$  A and*  
*atm-of ‘ (lits-of (trail S))  $\subseteq$  A*  
**shows** *atm-of ‘ (lits-of (trail T))  $\subseteq$  A*  
**using** *assms*  
**by** (*induction rule: cdcl<sub>NOT</sub>-all-induct*)  
(*simp-all add: dpll-bj-atms-in-trail-in-set dpll-bj-atms-of-m-clauses-inv*)

**lemma** *cdcl<sub>NOT</sub>-all-decomposition-implies*:

**assumes** *cdcl<sub>NOT</sub> S T and inv S and n-d[simp]: no-dup (trail S) and*  
*all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*  
**shows**  
*all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*  
**using** *assms(1,2,4)*

**proof** (*induction rule: cdcl<sub>NOT</sub>-all-induct*)

**case** *dpll-bj*  
**then show** *?case*  
**using** *dpll-bj-all-decomposition-implies-inv n-d by blast*

**next**

**case** *learn*  
**then show** *?case by (auto simp add: all-decomposition-implies-def)*

**next**

**case** (*forget<sub>NOT</sub> C T*) **note** *cls-C = this(1) and C = this(2) and T = this(3) and iniv = this(4)*  
**and**

*decomp = this(5)*

**show** *?case*

**unfolding** *all-decomposition-implies-def Ball-def*

**proof** (*intro allI, clarify*)

**fix** *a b*

**assume** *(a, b)  $\in$  set (get-all-marked-decomposition (trail T))*

**then have** *( $\lambda a. \{\#lit-of\ a\# \}$ ) ‘ set a  $\cup$  set-mset (clauses S)  $\models_{ps}$  ( $\lambda a. \{\#lit-of\ a\# \}$ ) ‘ set b*  
**using** *decomp T by (auto simp add: all-decomposition-implies-def)*

**moreover**

**have** *C  $\in$  set-mset (clauses S)*

**by** (*simp add: C*)

**then have** *set-mset (clauses T)  $\models_{ps}$  set-mset (clauses S)*

**by** (*metis (no-types) T clauses-remove-cls<sub>NOT</sub> cls-C insert-Diff order-refl*  
*set-mset-minus-replicate-mset(1) state-eq<sub>NOT</sub>-clauses true-clss-clss-def*  
*true-clss-clss-insert*)

**ultimately show** *( $\lambda a. \{\#lit-of\ a\# \}$ ) ‘ set a  $\cup$  set-mset (clauses T)*  
 $\models_{ps}$  *( $\lambda a. \{\#lit-of\ a\# \}$ ) ‘ set b*

**using** *true-clss-clss-generalise-true-clss-clss by blast*

**qed**

**qed**

**Extension of models** lemma *cdcl<sub>NOT</sub>-bj-sat-ext-iff*:

assumes *cdcl<sub>NOT</sub> S T* and *inv S* and *n-d: no-dup (trail S)*

shows  $I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$

using *assms*

**proof** (*induction rule:cdcl<sub>NOT</sub>-all-induct*)

case *dpll-bj*

then show *?case* by (*simp add: dpll-bj-clauses*)

**next**

case (*learn C T*) note  $T = \text{this}(3)$

{ fix *J*

assume

$I \models_{\text{sextm}} \text{clauses } S$  and

$I \subseteq J$  and

*tot: total-over-m J (set-mset ({#C#} + (clauses S)))* and

*cons: consistent-interp J*

then have  $J \models_{\text{sm}} \text{clauses } S$  **unfolding** *true-clss-ext-def* by *auto*

**moreover**

with  $\langle \text{clauses } S \models_{\text{pm}} C \rangle$  have  $J \models C$

using *tot cons* **unfolding** *true-clss-cls-def* by *auto*

ultimately have  $J \models_{\text{sm}} \{ \#C \# \} + \text{clauses } S$  by *auto*

}

then have  $H: I \models_{\text{sextm}} (\text{clauses } S) \implies I \models_{\text{sext}} \text{insert } C (\text{set-mset } (\text{clauses } S))$

**unfolding** *true-clss-ext-def* by *auto*

show *?case*

**apply** *standard*

using *T n-d* **apply** (*auto simp add: H*)[]

using *T n-d* **apply** *simp*

by (*metis Diff-insert-absorb insert-subset subsetI subset-antisym*  
*true-clss-ext-decrease-right-remove-r*)

**next**

case (*forget<sub>NOT</sub> C T*) note  $\text{cls-}C = \text{this}(1)$  and  $T = \text{this}(3)$

{ fix *J*

assume

$I \models_{\text{sext}} \text{set-mset } (\text{clauses } S) - \{C\}$  and

$I \subseteq J$  and

*tot: total-over-m J (set-mset (clauses S))* and

*cons: consistent-interp J*

then have  $J \models_{\text{s}} \text{set-mset } (\text{clauses } S) - \{C\}$

**unfolding** *true-clss-ext-def* by (*meson Diff-subset total-over-m-subset*)

**moreover**

with  $\text{cls-}C$  have  $J \models C$

using *tot cons* **unfolding** *true-clss-cls-def*

by (*metis Un-commute forget<sub>NOT</sub>.hyps(2) insert-Diff insert-is-Un mem-set-mset-iff order-refl*  
*set-mset-minus-replicate-mset(1)*)

ultimately have  $J \models_{\text{sm}} (\text{clauses } S)$  by (*metis insert-Diff-single true-clss-insert*)

}

then have  $H: I \models_{\text{sext}} \text{set-mset } (\text{clauses } S) - \{C\} \implies I \models_{\text{sextm}} (\text{clauses } S)$

**unfolding** *true-clss-ext-def* by *blast*

show *?case* using *T* by (*auto simp: true-clss-ext-decrease-right-remove-r H*)

**qed**

**end** — end of *conflict-driven-clause-learning-ops*

## 14.5 CDCL with invariant

```

locale conflict-driven-clause-learning =
  conflict-driven-clause-learning-ops +
  assumes  $cdcl_{NOT}\text{-inv}$ :  $\bigwedge S\ T. cdcl_{NOT}\ S\ T \implies inv\ S \implies inv\ T$ 
begin
sublocale dpll-with-backjumping
  apply unfold-locales
  using  $cdcl_{NOT}.simps\ cdcl_{NOT}\text{-inv}$  by auto

lemma rtrancpl-cdclNOT-inv:
   $cdcl_{NOT}^{**}\ S\ T \implies inv\ S \implies inv\ T$ 
  by (induction rule: rtrancpl-induct) (auto simp add:  $cdcl_{NOT}\text{-inv}$ )

lemma rtrancpl-cdclNOT-no-dup:
  assumes  $cdcl_{NOT}^{**}\ S\ T$  and  $inv\ S$ 
  and no-dup (trail  $S$ )
  shows no-dup (trail  $T$ )
  using assms by (induction rule: rtrancpl-induct) (auto intro:  $cdcl_{NOT}\text{-no-dup}\ rtrancpl\text{-}cdcl_{NOT}\text{-inv}$ )

lemma rtrancpl-cdclNOT-trail-clauses-bound:
  assumes
     $cdcl$ :  $cdcl_{NOT}^{**}\ S\ T$  and
     $inv$ :  $inv\ S$  and
     $n\text{-d}$ : no-dup (trail  $S$ ) and
     $atms\text{-clauses}\text{-}S$ :  $atms\text{-of}\ \mu\ (clauses\ S) \subseteq A$  and
     $atms\text{-trail}\text{-}S$ :  $atm\text{-of}\ (lits\text{-of}\ (trail\ S)) \subseteq A$ 
  shows  $atm\text{-of}\ (lits\text{-of}\ (trail\ T)) \subseteq A \wedge atms\text{-of}\ \mu\ (clauses\ T) \subseteq A$ 
  using  $cdcl$ 
proof (induction rule: rtrancpl-induct)
  case base
  then show ?case using  $atms\text{-clauses}\text{-}S\ atms\text{-trail}\text{-}S$  by simp
next
  case (step  $T\ U$ ) note  $st = this(1)$  and  $cdcl_{NOT} = this(2)$  and  $IH = this(3)$ 
  have  $inv\ T$  using  $inv\ st\ rtrancpl\text{-}cdcl_{NOT}\text{-inv}$  by blast
  have no-dup (trail  $T$ )
    using  $rtrancpl\text{-}cdcl_{NOT}\text{-no-dup}[of\ S\ T]\ st\ cdcl_{NOT}\ inv\ n\text{-d}$  by blast
  then have  $atms\text{-of}\ \mu\ (clauses\ U) \subseteq A$ 
    using  $cdcl_{NOT}\text{-}atms\text{-of}\ m\text{-clauses}\text{-decreasing}[OF\ cdcl_{NOT}]\ IH\ n\text{-d}\ (inv\ T)$  by auto
  moreover
    have  $atm\text{-of}\ (lits\text{-of}\ (trail\ U)) \subseteq A$ 
      using  $cdcl_{NOT}\text{-}atms\text{-in}\text{-trail}\text{-in}\text{-set}[OF\ cdcl_{NOT},\ of\ A]\ (no\text{-dup}\ (trail\ T))$ 
      by (meson  $atms\text{-trail}\text{-}S\ atms\text{-clauses}\text{-}S\ IH\ (inv\ T)\ cdcl_{NOT}$ )
    ultimately show ?case by fast
qed

lemma rtrancpl-cdclNOT-all-decomposition-implies:
  assumes  $cdcl_{NOT}^{**}\ S\ T$  and  $inv\ S$  and no-dup (trail  $S$ ) and
    all-decomposition-implies- $m\ (clauses\ S)\ (get\text{-all}\text{-marked}\text{-decomposition}\ (trail\ S))$ 
  shows
    all-decomposition-implies- $m\ (clauses\ T)\ (get\text{-all}\text{-marked}\text{-decomposition}\ (trail\ T))$ 
  using assms by (induction)
  (auto intro:  $rtrancpl\text{-}cdcl_{NOT}\text{-inv}\ cdcl_{NOT}\text{-all}\text{-decomposition}\text{-implies}\ rtrancpl\text{-}cdcl_{NOT}\text{-no-dup}$ )

lemma rtrancpl-cdclNOT-bj-sat-ext-iff:
  assumes  $cdcl_{NOT}^{**}\ S\ T$  and  $inv\ S$  and no-dup (trail  $S$ )

```

**shows**  $I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$   
**using** *assms* **apply** (induction rule: *rtrancpl-induct*)  
**using** *cdcl<sub>NOT</sub>-bj-sat-ext-iff* **by** (auto intro: *rtrancpl-cdcl<sub>NOT</sub>-inv* *rtrancpl-cdcl<sub>NOT</sub>-no-dup*)

**definition** *cdcl<sub>NOT</sub>-NOT-all-inv* **where**

*cdcl<sub>NOT</sub>-NOT-all-inv*  $A \ S \longleftrightarrow (\text{finite } A \wedge \text{inv } S \wedge \text{atms-of-mu } (\text{clauses } S) \subseteq \text{atms-of-m } A$   
 $\wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-m } A \wedge \text{no-dup } (\text{trail } S))$

**lemma** *cdcl<sub>NOT</sub>-NOT-all-inv*:

**assumes** *cdcl<sub>NOT</sub>\*\**  $S \ T$  **and** *cdcl<sub>NOT</sub>-NOT-all-inv*  $A \ S$

**shows** *cdcl<sub>NOT</sub>-NOT-all-inv*  $A \ T$

**using** *assms* **unfolding** *cdcl<sub>NOT</sub>-NOT-all-inv-def*

**by** (*simp* add: *rtrancpl-cdcl<sub>NOT</sub>-inv* *rtrancpl-cdcl<sub>NOT</sub>-no-dup* *rtrancpl-cdcl<sub>NOT</sub>-trail-clauses-bound*)

**abbreviation** *learn-or-forget* **where**

*learn-or-forget*  $S \ T \equiv (\lambda S \ T. \text{learn } S \ T \vee \text{forget}_{\text{NOT}} S \ T) \ S \ T$

**lemma** *rtrancpl-learn-or-forget-cdcl<sub>NOT</sub>*:

*learn-or-forget\*\**  $S \ T \implies \text{cdcl}_{\text{NOT}}^{**} S \ T$

**using** *rtrancpl-mono*[of *learn-or-forget cdcl<sub>NOT</sub>*] *cdcl<sub>NOT</sub>.c-learn* *cdcl<sub>NOT</sub>.c-forget<sub>NOT</sub>* **by** *blast*

**lemma** *learn-or-forget-dpll- $\mu_C$* :

**assumes**

*l-f*: *learn-or-forget\*\**  $S \ T$  **and**

*dpll*: *dpll-bj*  $T \ U$  **and**

*inv*: *cdcl<sub>NOT</sub>-NOT-all-inv*  $A \ S$

**shows**  $(2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))$

$- \mu_C (1 + \text{card } (\text{atms-of-m } A)) (2 + \text{card } (\text{atms-of-m } A)) (\text{trail-weight } U)$

$< (2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))$

$- \mu_C (1 + \text{card } (\text{atms-of-m } A)) (2 + \text{card } (\text{atms-of-m } A)) (\text{trail-weight } S)$

(is  $?_{\mu} U < ?_{\mu} S$ )

**proof** –

**have**  $?_{\mu} S = ?_{\mu} T$

**using** *l-f*

**proof** (*induction*)

**case** *base*

**then show**  $?_{\text{case}}$  **by** *simp*

**next**

**case** (*step*  $T \ U$ )

**moreover then have** *no-dup* (*trail*  $T$ )

**using** *rtrancpl-cdcl<sub>NOT</sub>-no-dup*[of  $S \ T$ ] *cdcl<sub>NOT</sub>-NOT-all-inv-def* *inv*

*rtrancpl-learn-or-forget-cdcl<sub>NOT</sub>* **by** *auto*

**ultimately show**  $?_{\text{case}}$

**using** *forget- $\mu_C$ -stable* *learn- $\mu_C$ -stable* *inv* **unfolding** *cdcl<sub>NOT</sub>-NOT-all-inv-def* **by** *presburger*

**qed**

**moreover have** *cdcl<sub>NOT</sub>-NOT-all-inv*  $A \ T$

**using** *rtrancpl-learn-or-forget-cdcl<sub>NOT</sub>* *cdcl<sub>NOT</sub>-NOT-all-inv* *l-f* *inv* **by** *blast*

**ultimately show**  $?_{\text{thesis}}$

**using** *dpll-bj-trail-mes-decreasing-prop*[of  $T \ U \ A$ , *OF* *dpll*] *finite*

**unfolding** *cdcl<sub>NOT</sub>-NOT-all-inv-def* **by** *linarith*

**qed**

**lemma** *infinite-cdcl<sub>NOT</sub>-exists-learn-and-forget-infinite-chain*:

**assumes**

```

 $\wedge i. \text{cdcl}_{NOT} (f i) (f (\text{Suc } i))$  and
 $\text{inv}: \text{cdcl}_{NOT-NOT-all-inv} A (f 0)$ 
shows  $\exists j. \forall i \geq j. \text{learn-or-forget} (f i) (f (\text{Suc } i))$ 
using assms
proof (induction ( $2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$ 
   $-\mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight} (f 0))$ 
  arbitrary: f
  rule: nat-less-induct-case)
case (Suc n) note  $IH = \text{this}(1)$  and  $\mu = \text{this}(2)$  and  $\text{cdcl}_{NOT} = \text{this}(3)$  and  $\text{inv} = \text{this}(4)$ 
consider
  (dpll-end)  $\exists j. \forall i \geq j. \text{learn-or-forget} (f i) (f (\text{Suc } i))$ 
  | (dpll-more)  $\neg(\exists j. \forall i \geq j. \text{learn-or-forget} (f i) (f (\text{Suc } i)))$ 
by blast
then show ?case
proof cases
  case dpll-end
    then show ?thesis by auto
next
  case dpll-more
    then have  $j: \exists i. \neg \text{learn} (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i))$ 
      by blast
    obtain i where
       $\neg \text{learn} (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i))$  and
       $\forall k < i. \text{learn-or-forget} (f k) (f (\text{Suc } k))$ 
    proof  $-$ 
      obtain  $i_0$  where  $\neg \text{learn} (f i_0) (f (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f i_0) (f (\text{Suc } i_0))$ 
        using j by auto
      then have  $\{i. i \leq i_0 \wedge \neg \text{learn} (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i))\} \neq \{\}$ 
        by auto
      let  $?I = \{i. i \leq i_0 \wedge \neg \text{learn} (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i))\}$ 
      let  $?i = \text{Min } ?I$ 
      have finite ?I
        by auto
      have  $\neg \text{learn} (f ?i) (f (\text{Suc } ?i)) \wedge \neg \text{forget}_{NOT} (f ?i) (f (\text{Suc } ?i))$ 
        using Min-in[OF (finite ?I) (?I ≠ {})] by auto
      moreover have  $\forall k < ?i. \text{learn-or-forget} (f k) (f (\text{Suc } k))$ 
        using Min.coboundedI[of  $\{i. i \leq i_0 \wedge \neg \text{learn} (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i))\}$ , simplified]
        by (meson  $(\neg \text{learn} (f i_0) (f (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f i_0) (f (\text{Suc } i_0)))$  less-imp-le
          dual-order.trans not-le)
      ultimately show ?thesis using that by blast
    qed
def  $g \equiv \lambda n. f (n + \text{Suc } i)$ 
have dpll-bj  $(f i) (g 0)$ 
  using  $(\neg \text{learn} (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i)))$  cdcl_{NOT} cdcl_{NOT}.cases
  g-def by auto
{
  fix j
  assume  $j \leq i$ 
  then have learn-or-forget**  $(f 0) (f j)$ 
    apply (induction j)
    apply simp
    by (metis (no-types, lifting) Suc-leD Suc-le-lessD rtrancIp.simps
       $\langle \forall k < i. \text{learn} (f k) (f (\text{Suc } k)) \vee \text{forget}_{NOT} (f k) (f (\text{Suc } k)) \rangle$ )
  }

```

**then have**  $\text{learn-or-forget}^{**} (f\ 0) (f\ i)$  **by** *blast*  
**then have**  $(2 + \text{card} (\text{atms-of-}m\ A)) \wedge (1 + \text{card} (\text{atms-of-}m\ A))$   
 $- \mu_C (1 + \text{card} (\text{atms-of-}m\ A)) (2 + \text{card} (\text{atms-of-}m\ A)) (\text{trail-weight} (g\ 0))$   
 $< (2 + \text{card} (\text{atms-of-}m\ A)) \wedge (1 + \text{card} (\text{atms-of-}m\ A))$   
 $- \mu_C (1 + \text{card} (\text{atms-of-}m\ A)) (2 + \text{card} (\text{atms-of-}m\ A)) (\text{trail-weight} (f\ 0))$   
**using**  $\text{learn-or-forget-dpll-}\mu_C[\text{of } f\ 0\ f\ i\ g\ 0\ A]$  *inv*  $\langle \text{dpll-bj} (f\ i) (g\ 0) \rangle$   
**unfolding**  $\text{cdcl}_{NOT}\text{-NOT-all-inv-def}$  **by** *linarith*  
  
**moreover have**  $\text{cdcl}_{NOT}\text{-}i: \text{cdcl}_{NOT}^{**} (f\ 0) (g\ 0)$   
**using**  $\text{rtrancpl-learn-or-forget-cdcl}_{NOT}[\text{of } f\ 0\ f\ i]$   $\langle \text{learn-or-forget}^{**} (f\ 0) (f\ i) \rangle$   
 $\text{cdcl}_{NOT}[\text{of } i]$  **unfolding**  $g\text{-def}$  **by** *auto*  
**moreover have**  $\bigwedge i. \text{cdcl}_{NOT} (g\ i) (g\ (\text{Suc } i))$   
**using**  $\text{cdcl}_{NOT} g\text{-def}$  **by** *auto*  
**moreover have**  $\text{cdcl}_{NOT}\text{-NOT-all-inv } A (g\ 0)$   
**using**  $\text{inv cdcl}_{NOT}\text{-}i\ \text{rtrancpl-cdcl}_{NOT}\text{-trail-clauses-bound } g\text{-def cdcl}_{NOT}\text{-NOT-all-inv}$  **by** *auto*  
**ultimately obtain**  $j$  **where**  $j: \bigwedge i. i \geq j \implies \text{learn-or-forget} (g\ i) (g\ (\text{Suc } i))$   
**using**  $IH$  **unfolding**  $\mu[\text{symmetric}]$  **by** *presburger*  
**show** *?thesis*  
**proof**  
 $\{$   
 $\text{fix } k$   
 $\text{assume } k \geq j + \text{Suc } i$   
**then have**  $\text{learn-or-forget} (f\ k) (f\ (\text{Suc } k))$   
**using**  $j[\text{of } k - \text{Suc } i]$  **unfolding**  $g\text{-def}$  **by** *auto*  
 $\}$   
**then show**  $\forall k \geq j + \text{Suc } i. \text{learn-or-forget} (f\ k) (f\ (\text{Suc } k))$   
**by** *auto*  
**qed**  
**qed**  
**next**  
**case**  $0$  **note**  $H = \text{this}(1)$  **and**  $\text{cdcl}_{NOT} = \text{this}(2)$  **and**  $\text{inv} = \text{this}(3)$   
**show** *?case*  
**proof** (*rule ccontr*)  
**assume**  $\neg ?\text{case}$   
**then have**  $j: \exists i. \neg \text{learn} (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))$   
**by** *blast*  
**obtain**  $i$  **where**  
 $\neg \text{learn} (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))$  **and**  
 $\forall k < i. \text{learn-or-forget} (f\ k) (f\ (\text{Suc } k))$   
**proof** –  
**obtain**  $i_0$  **where**  $\neg \text{learn} (f\ i_0) (f\ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f\ i_0) (f\ (\text{Suc } i_0))$   
**using**  $j$  **by** *auto*  
**then have**  $\{i. i \leq i_0 \wedge \neg \text{learn} (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))\} \neq \{\}$   
**by** *auto*  
**let**  $?I = \{i. i \leq i_0 \wedge \neg \text{learn} (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))\}$   
**let**  $?i = \text{Min } ?I$   
**have** *finite ?I*  
**by** *auto*  
**have**  $\neg \text{learn} (f\ ?i) (f\ (\text{Suc } ?i)) \wedge \neg \text{forget}_{NOT} (f\ ?i) (f\ (\text{Suc } ?i))$   
**using**  $\text{Min-in}[OF \langle \text{finite } ?I \rangle \langle ?I \neq \{\} \rangle]$  **by** *auto*  
**moreover have**  $\forall k < ?i. \text{learn-or-forget} (f\ k) (f\ (\text{Suc } k))$   
**using**  $\text{Min.coboundedI}[\text{of } \{i. i \leq i_0 \wedge \neg \text{learn} (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))\}, \text{simplified}]$   
**by** ( $\text{meson } (\neg \text{learn} (f\ i_0) (f\ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f\ i_0) (f\ (\text{Suc } i_0)))$ ) *less-imp-le*  
 $\text{dual-order.trans not-le}$

```

    ultimately show ?thesis using that by blast
  qed
have dpll-bj (f i) (f (Suc i))
  using (¬ learn (f i) (f (Suc i)) ∧ ¬ forgetNOT (f i) (f (Suc i))) cdclNOT cdclNOT.cases
  by blast
{
  fix j
  assume j ≤ i
  then have learn-or-forget** (f 0) (f j)
    apply (induction j)
    apply simp
    by (metis (no-types, lifting) Suc-leD Suc-le-lessD rtranclp.simps
      ⟨∀ k < i. learn (f k) (f (Suc k)) ∨ forgetNOT (f k) (f (Suc k))⟩)
}
then have learn-or-forget** (f 0) (f i) by blast

then show False
  using learn-or-forget-dpll-μC[of f 0 f i f (Suc i) A] inv 0
  ⟨dpll-bj (f i) (f (Suc i))⟩ unfolding cdclNOT-NOT-all-inv-def by linarith
qed
qed

lemma wf-cdclNOT-no-learn-and-forget-infinite-chain:
  assumes
    no-infinite-lf: ∧ f j. ¬ (∀ i ≥ j. learn-or-forget (f i) (f (Suc i)))
  shows wf {(T, S). cdclNOT S T ∧ cdclNOT-NOT-all-inv A S} (is wf {(T, S). cdclNOT S T
    ∧ ?inv S})
  unfolding wf-iff-no-infinite-down-chain
proof (rule ccontr)
  assume ¬ ¬ (∃ f. ∀ i. (f (Suc i), f i) ∈ {(T, S). cdclNOT S T ∧ ?inv S})
  then obtain f where
    ∀ i. cdclNOT (f i) (f (Suc i)) ∧ ?inv (f i)
  by fast
  then have ∃ j. ∀ i ≥ j. learn-or-forget (f i) (f (Suc i))
    using infinite-cdclNOT-exists-learn-and-forget-infinite-chain[of f] by meson
  then show False using no-infinite-lf by blast
qed

lemma inv-and-tranclp-cdclNOT-tranclp-cdclNOT-and-inv:
  cdclNOT++ S T ∧ cdclNOT-NOT-all-inv A S ⟷ (λ S T. cdclNOT S T ∧ cdclNOT-NOT-all-inv A
  S)++ S T
  (is ?A ∧ ?I ⟷ ?B)
proof
  assume ?A ∧ ?I
  then have ?A and ?I by blast+
  then show ?B
    apply induction
    apply (simp add: tranclp.r-into-trancl)
    by (metis (no-types, lifting) cdclNOT-NOT-all-inv tranclp.simps tranclp-into-rtranclp)
next
  assume ?B
  then have ?A by induction auto
  moreover have ?I using ⟨?B⟩ tranclpD by fastforce
  ultimately show ?A ∧ ?I by blast
qed

```

**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*  $\{(T, S). \text{cdcl}_{NOT}^{++} S T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A S\}$

**using** *wf-trancl*[*OF wf-cdcl<sub>NOT</sub>-no-learn-and-forget-infinite-chain*[*OF no-infinite-lf*]]

**apply** (*rule wf-subset*)

**by** (*auto simp: trancl-set-tranclp inv-and-tranclp-cdcl<sub>NOT</sub>-tranclp-cdcl<sub>NOT</sub>-and-inv*)

**lemma** *cdcl<sub>NOT</sub>-final-state*:

**assumes**

*n-s*: *no-step cdcl<sub>NOT</sub> S* **and**

*inv*: *cdcl<sub>NOT</sub>-NOT-all-inv A S* **and**

*decomp*: *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

**shows** *unsatisfiable (set-mset (clauses S))*

$\vee (\text{trail } S \models_{asm} \text{clauses } S \wedge \text{satisfiable } (\text{set-mset } (\text{clauses } S)))$

**proof** –

**have** *n-s'*: *no-step dpll-bj S*

**using** *n-s* **by** (*auto simp: cdcl<sub>NOT</sub>.simps*)

**show** *?thesis*

**apply** (*rule dpll-backjump-final-state*[*of S A*])

**using** *inv decomp n-s'* **unfolding** *cdcl<sub>NOT</sub>-NOT-all-inv-def* **by** *auto*

**qed**

**lemma** *full-cdcl<sub>NOT</sub>-final-state*:

**assumes**

*full*: *full cdcl<sub>NOT</sub> S T* **and**

*inv*: *cdcl<sub>NOT</sub>-NOT-all-inv A S* **and**

*n-d*: *no-dup (trail S)* **and**

*decomp*: *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

**shows** *unsatisfiable (set-mset (clauses T))*

$\vee (\text{trail } T \models_{asm} \text{clauses } T \wedge \text{satisfiable } (\text{set-mset } (\text{clauses } T)))$

**proof** –

**have** *st*: *cdcl<sub>NOT</sub>\*\* S T* **and** *n-s*: *no-step cdcl<sub>NOT</sub> T*

**using** *full* **unfolding** *full-def* **by** *blast+*

**have** *n-s'*: *cdcl<sub>NOT</sub>-NOT-all-inv A T*

**using** *cdcl<sub>NOT</sub>-NOT-all-inv inv st* **by** *blast*

**moreover have** *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*

**using** *cdcl<sub>NOT</sub>-NOT-all-inv-def decomp inv rtranclp-cdcl<sub>NOT</sub>-all-decomposition-implies st* **by** *auto*

**ultimately show** *?thesis*

**using** *cdcl<sub>NOT</sub>-final-state n-s* **by** *blast*

**qed**

**end** — end of *conflict-driven-clause-learning*

## 14.6 Termination

### 14.6.1 Restricting learn and forget

**locale** *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =

*conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub>*

*propagate-conds inv backjump-conds*

$\lambda C S. \text{distinct-mset } C \wedge \neg \text{tautology } C \wedge \text{learn-restrictions } C S \wedge$

$(\exists F K d F' C' L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge C = C' + \{\#L\} \wedge F \models_{as} C \text{Not } C'$

$\wedge C' + \{\#L\} \notin \# \text{clauses } S)$

$\lambda C S. \neg (\exists F' F K d L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } (C - \{\#L\}))$



```

 $\wedge$  forget-restrictions  $C$   $S$ 
for
  trail :: 'st  $\Rightarrow$  ('v::linorder, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  learn-restrictions forget-restrictions :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

lemma cdclNOT-learn-all-induct[consumes 1, case-names dpll-bj learn forgetNOT]:
  fixes  $S$   $T$  :: 'st
  assumes cdclNOT  $S$   $T$  and
  dpll:  $\bigwedge T. \text{dpll-bj } S \ T \Longrightarrow P \ S \ T$  and
  learning:
     $\bigwedge C \ F \ K \ F' \ C' \ L \ T. \text{clauses } S \models_{pm} C$ 
     $\Longrightarrow \text{atms-of } C \subseteq \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$ 
     $\Longrightarrow \text{distinct-mset } C \Longrightarrow \neg \text{tautology } C \Longrightarrow \text{learn-restrictions } C \ S$ 
     $\Longrightarrow \text{trail } S = F' @ \text{Marked } K \ () \ \# \ F \Longrightarrow C = C' + \{\#L\# \} \Longrightarrow F \models_{as} C \text{Not } C'$ 
     $\Longrightarrow C' + \{\#L\# \} \notin \text{clauses } S \Longrightarrow T \sim \text{add-cl}_{NOT} \ C \ S$ 
     $\Longrightarrow P \ S \ T$  and
  forgetting:  $\bigwedge C \ T. \text{clauses } S - \text{replicate-mset (count (clauses } S) \ C) \ C \models_{pm} C$ 
     $\Longrightarrow C \in \# \text{clauses } S$ 
     $\Longrightarrow \neg(\exists F' \ F \ K \ L. \text{trail } S = F' @ \text{Marked } K \ () \ \# \ F \wedge F \models_{as} C \text{Not } (C - \{\#L\# \}))$ 
     $\Longrightarrow T \sim \text{remove-cl}_{NOT} \ C \ S$ 
     $\Longrightarrow \text{forget-restrictions } C \ S \Longrightarrow P \ S \ T$ 
  shows  $P \ S \ T$ 
  using assms(1)
  apply (induction rule: cdclNOT.induct)
  apply (auto dest: assms(2) simp add: learn-ops-axioms)[]
  apply (auto elim!: learn-ops.learn.cases[OF learn-ops-axioms] dest: assms(3))[]
  apply (auto elim!: forget-ops.forgetNOT.cases[OF forget-ops-axioms] dest!: assms(4))
  done

lemma rtranclp-cdclNOT-inv:
  cdclNOT**  $S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 
  apply (induction rule: rtranclp-induct)
  apply simp
  using cdclNOT-inv unfolding conflict-driven-clause-learning-def
  conflict-driven-clause-learning-axioms-def by blast

lemma learn-always-simple-clauses:
  assumes
    learn: learn  $S \ T$  and
    n-d: no-dup (trail  $S$ )
  shows set-mset (clauses  $T$  - clauses  $S$ )
     $\subseteq \text{build-all-simple-clss (atms-of-mu (clauses } S) \cup \text{atm-of ' lits-of (trail } S))$ 
proof
  fix  $C$  assume  $C$ :  $C \in \text{set-mset (clauses } T - \text{clauses } S)$ 
  have distinct-mset  $C \neg \text{tautology } C$  using learn  $C$  n-d by (elim learnE; auto)+
  then have  $C \in \text{build-all-simple-clss (atms-of } C)$ 
  using distinct-mset-not-tautology-implies-in-build-all-simple-clss by blast

```

**moreover have**  $\text{atms-of } C \subseteq \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S)$   
**using**  $\text{learn } C \text{ n-d by } (\text{elim learnE}) (\text{auto simp: atms-of-m-def atms-of-def image-Un true-annots-CNot-all-atms-defined})$   
**moreover have**  $\text{finite } (\text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S))$   
**by**  $\text{auto}$   
**ultimately show**  $C \in \text{build-all-simple-clss } (\text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S))$   
**using**  $\text{build-all-simple-clss-mono by } (\text{metis } (\text{no-types}) \text{ insert-subset mk-disjoint-insert})$   
**qed**

**definition**  $\text{conflicting-bj-clss } S \equiv$   
 $\{C + \{\#L\# \} \mid C \text{ L. } C + \{\#L\# \} \in \# \text{ clauses } S \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$   
 $\wedge (\exists F' K F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} \text{CNot } C)\}$

**lemma**  $\text{conflicting-bj-clss-remove-clss}_{\text{NOT}}[\text{simp}]$ :  
 $\text{conflicting-bj-clss } (\text{remove-clss}_{\text{NOT}} C S) = \text{conflicting-bj-clss } S - \{C\}$   
**unfolding**  $\text{conflicting-bj-clss-def by fastforce}$

**lemma**  $\text{conflicting-bj-clss-add-clss}_{\text{NOT}}\text{-state-eq}$ :  
 $T \sim \text{add-clss}_{\text{NOT}} C' S \implies \text{no-dup } (\text{trail } S) \implies \text{conflicting-bj-clss } T$   
 $= \text{conflicting-bj-clss } S$   
 $\cup (\text{if } \exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$   
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} \text{CNot } C)$   
 $\text{then } \{C'\} \text{ else } \{\})$   
**unfolding**  $\text{conflicting-bj-clss-def by auto metis+}$

**lemma**  $\text{conflicting-bj-clss-add-clss}_{\text{NOT}}$ :  
 $\text{no-dup } (\text{trail } S) \implies$   
 $\text{conflicting-bj-clss } (\text{add-clss}_{\text{NOT}} C' S)$   
 $= \text{conflicting-bj-clss } S$   
 $\cup (\text{if } \exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$   
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} \text{CNot } C)$   
 $\text{then } \{C'\} \text{ else } \{\})$   
**using**  $\text{conflicting-bj-clss-add-clss}_{\text{NOT}}\text{-state-eq by auto}$

**lemma**  $\text{conflicting-bj-clss-incl-clauses}$ :  
 $\text{conflicting-bj-clss } S \subseteq \text{set-mset } (\text{clauses } S)$   
**unfolding**  $\text{conflicting-bj-clss-def by auto}$

**lemma**  $\text{finite-conflicting-bj-clss}[\text{simp}]$ :  
 $\text{finite } (\text{conflicting-bj-clss } S)$   
**using**  $\text{conflicting-bj-clss-incl-clauses[of } S] \text{ rev-finite-subset by blast}$

**lemma**  $\text{learn-conflicting-increasing}$ :  
 $\text{no-dup } (\text{trail } S) \implies \text{learn } S T \implies \text{conflicting-bj-clss } S \subseteq \text{conflicting-bj-clss } T$   
**apply**  $(\text{elim learnE})$   
**by**  $(\text{subst } \text{conflicting-bj-clss-add-clss}_{\text{NOT}}\text{-state-eq[of } T]) \text{ auto}$

**abbreviation**  $\text{conflicting-bj-clss-yet } b S \equiv$   
 $3 \wedge b - \text{card } (\text{conflicting-bj-clss } S)$

**abbreviation**  $\mu_L :: \text{nat} \Rightarrow 'st \Rightarrow \text{nat} \times \text{nat}$  **where**  
 $\mu_L b S \equiv (\text{conflicting-bj-clss-yet } b S, \text{card } (\text{set-mset } (\text{clauses } S)))$

**lemma**  $\text{do-not-forget-before-backtrack-rule-clause-learned-clause-untouched}$ :  
**assumes**  $\text{forget}_{\text{NOT}} S T$

**shows** *conflicting-bj-clss*  $S = \text{conflicting-bj-clss } T$   
**using** *assms* **apply** *induction*  
**unfolding** *conflicting-bj-clss-def*  
**by** (*metis* (*no-types*, *lifting*) *Diff-insert-absorb* *Set.set-insert* *clauses-remove-clss<sub>NOT</sub>*  
*diff-union-cancelR* *insert-iff* *mem-set-mset-iff* *order-refl* *set-mset-minus-replicate-mset*(1)  
*state-eq<sub>NOT</sub>-clauses* *state-eq<sub>NOT</sub>-trail* *trail-remove-clss<sub>NOT</sub>*)

**lemma** *forget- $\mu_L$ -decrease*:  
**assumes** *forget<sub>NOT</sub>*: *forget<sub>NOT</sub> S T*  
**shows**  $(\mu_L \ b \ T, \mu_L \ b \ S) \in \text{less-than} \text{ } <*\text{lex}*> \text{ less-than}$   
**proof** –  
**have** *card* (*set-mset* (*clauses T*)) < *card* (*set-mset* (*clauses S*))  
**using** *forget<sub>NOT</sub>* **apply** *induction*  
**by** (*metis* *card-Diff1-less* *clauses-remove-clss<sub>NOT</sub>* *finite-set-mset* *mem-set-mset-iff* *order-refl*  
*set-mset-minus-replicate-mset*(1) *state-eq<sub>NOT</sub>-clauses*)  
**then show** ?thesis  
**unfolding** *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*[*OF forget<sub>NOT</sub>*]  
**by** *auto*  
**qed**

**lemma** *set-condition-or-split*:  
 $\{a. (a = b \vee Q \ a) \wedge S \ a\} = (\text{if } S \ b \text{ then } \{b\} \text{ else } \{\}) \cup \{a. Q \ a \wedge S \ a\}$   
**by** *auto*

**lemma** *set-insert-neq*:  
 $A \neq \text{insert } a \ A \longleftrightarrow a \notin A$   
**by** *auto*

**lemma** *learn- $\mu_L$ -decrease*:  
**assumes** *learnST*: *learn S T* **and** *n-d*: *no-dup* (*trail S*) **and**  
*A*: *atms-of-mu* (*clauses S*)  $\cup$  *atm-of* ‘*lits-of* (*trail S*)’  $\subseteq A$  **and**  
*fin-A*: *finite A*  
**shows**  $(\mu_L \ (\text{card } A) \ T, \mu_L \ (\text{card } A) \ S) \in \text{less-than} \text{ } <*\text{lex}*> \text{ less-than}$   
**proof** –  
**have** [*simp*]: (*atms-of-mu* (*clauses T*)  $\cup$  *atm-of* ‘*lits-of* (*trail T*)’)  
 $=$  (*atms-of-mu* (*clauses S*)  $\cup$  *atm-of* ‘*lits-of* (*trail S*)’)  
**using** *learnST* *n-d* **by** (*elim learnE*) *auto*  
**then have** *card* (*atms-of-mu* (*clauses T*)  $\cup$  *atm-of* ‘*lits-of* (*trail T*)’)  
 $=$  *card* (*atms-of-mu* (*clauses S*)  $\cup$  *atm-of* ‘*lits-of* (*trail S*)’)  
**by** (*auto intro!*: *card-mono*)  
**then have**  $\exists: (\exists::\text{nat}) \wedge \text{card} \ (\text{atms-of-mu} \ (\text{clauses } T) \cup \text{atm-of} \ \text{'lits-of' } (\text{trail } T))$   
 $= \exists \wedge \text{card} \ (\text{atms-of-mu} \ (\text{clauses } S) \cup \text{atm-of} \ \text{'lits-of' } (\text{trail } S))$   
**by** (*auto intro*: *power-mono*)  
**moreover have** *conflicting-bj-clss S*  $\subseteq$  *conflicting-bj-clss T*  
**using** *learnST* *n-d* **by** (*simp add*: *learn-conflicting-increasing*)  
**moreover have** *conflicting-bj-clss S*  $\neq$  *conflicting-bj-clss T*  
**using** *learnST*  
**proof** (*elim learnE*, *goal-cases*)  
**case** (1 *C*) **note** *clss-S* = *this*(1) **and** *atms-C* = *this*(2) **and** *inv* = *this*(3) **and** *T* = *this*(4)  
**then obtain** *F K F' C' L* **where**  
*tr-S*: *trail S* = *F' @ Marked K* () # *F* **and**  
*C*: *C* = *C' + {#L#}* **and**  
*F*: *F*  $\models_{\text{as}} C \text{Not } C'$  **and**  
*C-S*: *C' + {#L#}*  $\notin \#$  *clauses S*

```

  by blast
  moreover have distinct-mset  $C \neg \text{tautology } C$  using inv by blast+
  ultimately have  $C' + \{\#L\# \} \in \text{conflicting-bj-clss } T$ 
    using  $T \text{ n-d unfolding conflicting-bj-clss-def}$  by fastforce
  moreover have  $C' + \{\#L\# \} \notin \text{conflicting-bj-clss } S$ 
    using  $C\text{-}S \text{ unfolding conflicting-bj-clss-def}$  by auto
  ultimately show ?case by blast
qed
moreover have fin-T: finite (conflicting-bj-clss  $T$ )
  using learnST by induction (auto simp add: conflicting-bj-clss-add-clssNOT)
ultimately have card (conflicting-bj-clss  $T$ )  $\geq$  card (conflicting-bj-clss  $S$ )
  using card-mono by blast

moreover
  have fin': finite (atms-of-mu (clauses  $T$ )  $\cup$  atm-of ' lits-of (trail  $T$ ) )
    by auto
  have 1:atms-of-m (conflicting-bj-clss  $T$ )  $\subseteq$  atms-of-mu (clauses  $T$ )
    unfolding conflicting-bj-clss-def atms-of-m-def by auto
  have 2:  $\bigwedge x. x \in \text{conflicting-bj-clss } T \implies \neg \text{tautology } x \wedge \text{distinct-mset } x$ 
    unfolding conflicting-bj-clss-def by auto
  have  $T$ : conflicting-bj-clss  $T$ 
 $\subseteq$  build-all-simple-clss (atms-of-mu (clauses  $T$ )  $\cup$  atm-of ' lits-of (trail  $T$ ) )
    by standard (meson 1 2 fin' ' finite (conflicting-bj-clss  $T$ ) ' build-all-simple-clss-mono
      distinct-mset-set-def simplified-in-build-all subsetCE sup.coboundedI1)
moreover
  then have  $\#$ :  $3 \wedge \text{card} (\text{atms-of-mu} (\text{clauses } T) \cup \text{atm-of ' lits-of (trail } T))$ 
     $\geq \text{card} (\text{conflicting-bj-clss } T)$ 
    by (meson Nat.le-trans build-all-simple-clss-card build-all-simple-clss-finite card-mono fin')
  have atms-of-mu (clauses  $T$ )  $\cup$  atm-of ' lits-of (trail  $T$ )  $\subseteq A$ 
    using learnE[OF learnST]  $A$  by simp
  then have  $3 \wedge (\text{card } A) \geq \text{card} (\text{conflicting-bj-clss } T)$ 
    using  $\#$  fin-A by (meson build-all-simple-clss-card build-all-simple-clss-finite
      build-all-simple-clss-mono calculation(2) card-mono dual-order.trans)
  ultimately show ?thesis
    using psubset-card-mono[OF fin-T]
    unfolding less-than-iff lex-prod-def by clarify
    (meson ' conflicting-bj-clss  $S \neq \text{conflicting-bj-clss } T$  '
      ' conflicting-bj-clss  $S \subseteq \text{conflicting-bj-clss } T$  '
      diff-less-mono2 le-less-trans not-le psubsetI)
qed

```

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* (*trail*  $S$ )  $\subseteq$  *atms-of-m*  $A$  and in the clauses *atms-of-mu* (*clauses*  $S$ )  $\subseteq$  *atms-of-m*  $A$ . 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**

$$\mu_{CDCL} A \ T \equiv ((2 + \text{card} (\text{atms-of-m } A)) \wedge (1 + \text{card} (\text{atms-of-m } A)) \\ - \mu_C (1 + \text{card} (\text{atms-of-m } A)) (2 + \text{card} (\text{atms-of-m } A)) (\text{trail-weight } T), \\ \text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-m } A)) \ T, \text{card} (\text{set-mset} (\text{clauses } T)))$$

**lemma** *cdcl<sub>NOT</sub>-decreasing-measure*:  
**assumes**  
*cdcl<sub>NOT</sub> S T and*  
*inv: inv S and*  
*atm-clss: atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and*  
*atm-lits: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-m A and*  
*n-d: no-dup (trail S) and*  
*fin-A: finite A*  
**shows** ( $\mu_{CDCL} A T, \mu_{CDCL} A S$ )  
 $\in$  *less-than <\*lex\*> (less-than <\*lex\*> less-than)*  
**using** *assms(1)*  
**proof** *induction*  
**case** (*c-dpll-bj T*)  
**from** *dpll-bj-trail-mes-decreasing-prop[OF this(1) inv atm-clss atm-lits n-d fin-A]*  
**show** ?*case* **unfolding**  $\mu_{CDCL}$ -*def*  
**by** (*meson in-lex-prod less-than-iff*)  
**next**  
**case** (*c-learn T*) **note** *learn = this(1)*  
**then have** *S: trail S = trail T*  
**using** *inv atm-clss atm-lits n-d fin-A*  
**by** (*elim learnE*) *auto*  
**show** ?*case*  
**using** *learn- $\mu_L$ -decrease[OF learn - ] atm-clss atm-lits fin-A n-d* **unfolding**  $\mu_{CDCL}$ -*def* **by** *auto*  
**next**  
**case** (*c-forget<sub>NOT</sub> T*) **note** *forget<sub>NOT</sub> = this(1)*  
**have** *trail S = trail T* **using** *forget<sub>NOT</sub>* **by** *induction auto*  
**then show** ?*case*  
**using** *forget- $\mu_L$ -decrease[OF forget<sub>NOT</sub>]* **unfolding**  $\mu_{CDCL}$ -*def* **by** *auto*  
**qed**

**lemma** *wf-cdcl<sub>NOT</sub>-restricted-learning*:  
**assumes** *finite A*  
**shows** *wf { (T, S).*  
*(atms-of-mu (clauses S)  $\subseteq$  atms-of-m A  $\wedge$  atm-of ' lits-of (trail S)  $\subseteq$  atms-of-m A*  
 *$\wedge$  no-dup (trail S)*  
 *$\wedge$  inv S)*  
 *$\wedge$  cdcl<sub>NOT</sub> S T }*  
**by** (*rule wf-wf-if-measure'[of less-than <\*lex\*> (less-than <\*lex\*> less-than)]*)  
*(auto intro: cdcl<sub>NOT</sub>-decreasing-measure[OF - - - - assms])*

**definition**  $\mu_C' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$  **where**  
 $\mu_C' A T \equiv \mu_C (1 + \text{card (atms-of-m A)}) (2 + \text{card (atms-of-m A)}) (\text{trail-weight } T)$

**definition**  $\mu_{CDCL}' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$  **where**  
 $\mu_{CDCL}' A T \equiv$   
 $((2 + \text{card (atms-of-m A)}) \wedge (1 + \text{card (atms-of-m A)}) - \mu_C' A T) * (1 + 3^{\text{card (atms-of-m A)}}) * 2$   
 $+ \text{conflicting-bj-clss-yet (card (atms-of-m A)) } T * 2$   
 $+ \text{card (set-mset (clauses T))}$

**lemma** *cdcl<sub>NOT</sub>-decreasing-measure'*:  
**assumes**  
*cdcl<sub>NOT</sub> S T and*  
*inv: inv S and*  
*atms-clss: atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and*  
*atms-trail: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-m A and*

$n\text{-d}$ : *no-dup* (*trail*  $S$ ) **and**  
 $\text{fin-}A$ : *finite*  $A$   
**shows**  $\mu_{CDCL}' A T < \mu_{CDCL}' A S$   
**using** *assms*(1)  
**proof** (*induction rule*: *cdcl<sub>NOT</sub>-learn-all-induct*)  
**case** (*dpll-bj*  $T$ )  
**then have**  $(2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A T$   
 $< (2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A S$   
**using** *dpll-bj-trail-mes-decreasing-prop* *fin-}A* *inv*  $n\text{-d}$  *atms-clss* *atms-trail*  
**unfolding**  $\mu_C'$ -*def* **by** *blast*  
**then have**  $XX: ((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A T) + 1$   
 $\leq (2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A S$   
**by** *auto*  
**from** *mult-le-mono1*[*OF this, of*  $(1 + 3 \wedge \text{card } (\text{atms-of-}m A))$ ]  
**have**  $((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A T) *$   
 $(1 + 3 \wedge \text{card } (\text{atms-of-}m A)) + (1 + 3 \wedge \text{card } (\text{atms-of-}m A))$   
 $\leq ((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A S)$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-}m A))$   
**unfolding** *Nat.add-mult-distrib*  
**by** *presburger*  
**moreover**  
**have** *cl-T-S*: *clauses*  $T = \text{clauses } S$   
**using** *dpll-bj.hyps* *inv* *dpll-bj-clauses* **by** *auto*  
**have** *conflicting-bj-clss-yet* (*card* (*atms-of-}m A*))  $S < 1 + 3 \wedge \text{card } (\text{atms-of-}m A)$   
**by** *simp*  
**ultimately have**  $((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A T)$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-}m A)) + \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m A)) T$   
 $< ((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A S) * (1 + 3 \wedge \text{card } (\text{atms-of-}m A))$   
 $A))$   
**by** *linarith*  
**then have**  $((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A T)$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-}m A))$   
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m A)) T$   
 $< ((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A S)$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-}m A))$   
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m A)) S$   
**by** *linarith*  
**then have**  $((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A T)$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-}m A)) * 2$   
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m A)) T * 2$   
 $< ((2 + \text{card } (\text{atms-of-}m A)) \wedge (1 + \text{card } (\text{atms-of-}m A)) - \mu_C' A S)$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-}m A)) * 2$   
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m A)) S * 2$   
**by** *linarith*  
**then show** ?*case* **unfolding**  $\mu_{CDCL}'$ -*def* *cl-T-S* **by** *presburger*  
**next**  
**case** (*learn*  $C F' K F C' L T$ ) **note** *clss-S-C* = *this*(1) **and** *atms-C* = *this*(2) **and** *dist* = *this*(3)  
**and** *tauto* = *this*(4) **and** *learn-restr* = *this*(5) **and** *tr-S* = *this*(6) **and**  $C' = \text{this}(7)$  **and**  
 $F\text{-}C = \text{this}(8)$  **and**  $C\text{-new} = \text{this}(9)$  **and**  $T = \text{this}(10)$   
**have** *insert*  $C$  (*conflicting-bj-clss*  $S$ )  $\subseteq \text{build-all-simple-clss } (\text{atms-of-}m A)$   
**proof** –  
**have**  $C \in \text{build-all-simple-clss } (\text{atms-of-}m A)$   
**by** (*metis* (*no-types*, *hide-lams*) *Un-subset-iff atms-of-m-finite build-all-simple-clss-mono*  
*contra-subsetD dist distinct-mset-not-tautology-implies-in-build-all-simple-clss*  
*dual-order.trans fin-}A* *atms-C* *atms-clss* *atms-trail* *tauto*)

```

moreover have conflicting-bj-clss  $S \subseteq \text{build-all-simple-clss } (\text{atms-of-}m \ A)$ 
unfolding conflicting-bj-clss-def
proof
  fix  $x :: 'v \text{ literal multiset}$ 
  assume  $x \in \{C + \{\#L\# \} \mid C \ L. \ C + \{\#L\# \} \in \# \text{ clauses } S$ 
     $\wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$ 
     $\wedge (\exists F' \ K \ F. \text{trail } S = F' @ \text{Marked } K \ () \ \# \ F \wedge F \models_{as} C \text{Not } C)\}$ 
  then have  $\exists m \ l. \ x = m + \{\#l\# \} \wedge m + \{\#l\# \} \in \# \text{ clauses } S$ 
     $\wedge \text{distinct-mset } (m + \{\#l\# \}) \wedge \neg \text{tautology } (m + \{\#l\# \})$ 
     $\wedge (\exists ms \ l \ msa. \text{trail } S = ms @ \text{Marked } l \ () \ \# \ msa \wedge msa \models_{as} C \text{Not } m)$ 
  by blast
  then show  $x \in \text{build-all-simple-clss } (\text{atms-of-}m \ A)$ 
  by (meson atms-clss atms-of-atms-of-m-mono atms-of-m-finite build-all-simple-clss-mono
    distinct-mset-not-tautology-implies-in-build-all-simple-clss fin-A finite-subset
    mem-set-mset-iff set-rev-mp)
  qed
ultimately show ?thesis
by auto
qed
then have  $\text{card } (\text{insert } C \ (\text{conflicting-bj-clss } S)) \leq 3 \wedge (\text{card } (\text{atms-of-}m \ A))$ 
by (meson Nat.le-trans atms-of-m-finite build-all-simple-clss-card build-all-simple-clss-finite
  card-mono fin-A)
moreover have [simp]:  $\text{card } (\text{insert } C \ (\text{conflicting-bj-clss } S))$ 
   $= \text{Suc } (\text{card } ((\text{conflicting-bj-clss } S)))$ 
by (metis (no-types) C' C-new card-insert-if conflicting-bj-clss-incl-clauses contra-subsetD
  finite-conflicting-bj-clss mem-set-mset-iff)
moreover have [simp]:  $\text{conflicting-bj-clss } (\text{add-cl}_{NOT} \ C \ S) = \text{conflicting-bj-clss } S \cup \{C\}$ 
using dist tauto F-C n-d by (subst conflicting-bj-clss-add-cl_{NOT})
  (force simp add: ac-simps C' tr-S) +
ultimately have [simp]:  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m \ A)) \ S$ 
   $= \text{Suc } (\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m \ A)) \ (\text{add-cl}_{NOT} \ C \ S))$ 
by simp
have 1:  $\text{clauses } T = \text{clauses } (\text{add-cl}_{NOT} \ C \ S)$  using  $T$  by auto
have 2:  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m \ A)) \ T$ 
   $= \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m \ A)) \ (\text{add-cl}_{NOT} \ C \ S)$ 
using  $T$  unfolding conflicting-bj-clss-def by auto
have 3:  $\mu_{C'} \ A \ T = \mu_{C'} \ A \ (\text{add-cl}_{NOT} \ C \ S)$ 
using  $T$  unfolding  $\mu_{C'}$ -def by auto
have  $((2 + \text{card } (\text{atms-of-}m \ A)) \wedge (1 + \text{card } (\text{atms-of-}m \ A)) - \mu_{C'} \ A \ (\text{add-cl}_{NOT} \ C \ S))$ 
   $* (1 + 3 \wedge \text{card } (\text{atms-of-}m \ A)) * 2$ 
   $= ((2 + \text{card } (\text{atms-of-}m \ A)) \wedge (1 + \text{card } (\text{atms-of-}m \ A)) - \mu_{C'} \ A \ S)$ 
   $* (1 + 3 \wedge \text{card } (\text{atms-of-}m \ A)) * 2$ 
using n-d unfolding  $\mu_{C'}$ -def by auto
moreover
have  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m \ A)) \ (\text{add-cl}_{NOT} \ C \ S)$ 
   $* 2$ 
   $+ \text{card } (\text{set-mset } (\text{clauses } (\text{add-cl}_{NOT} \ C \ S)))$ 
   $< \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-}m \ A)) \ S * 2$ 
   $+ \text{card } (\text{set-mset } (\text{clauses } S))$ 
by (simp add: C' C-new n-d)
ultimately show ?case unfolding  $\mu_{CDCL}$ '-def 1 2 3 by presburger
next
case (forget_{NOT} C T) note  $T = \text{this}(4)$ 
have [simp]:  $\mu_{C'} \ A \ (\text{remove-cl}_{NOT} \ C \ S) = \mu_{C'} \ A \ S$ 
unfolding  $\mu_{C'}$ -def by auto

```

**have**  $\text{forget}_{NOT} S T$   
**apply** ( $\text{rule } \text{forget}_{NOT}.\text{intros}$ ) **using**  $\text{forget}_{NOT}$  **by**  $\text{auto}$   
**then have**  $\text{conflicting-bj-clss } T = \text{conflicting-bj-clss } S$   
**using**  $\text{do-not-forget-before-backtrack-rule-clause-learned-clause-untouched}$  **by**  $\text{blast}$   
**moreover have**  $\text{card } (\text{set-mset } (\text{clauses } T)) < \text{card } (\text{set-mset } (\text{clauses } S))$   
**by** ( $\text{metis } T \text{ card-Diff1-less clauses-remove-cl}_{NOT} \text{ finite-set-mset forget}_{NOT}.\text{hyps}(2)$   
 $\text{mem-set-mset-iff order-refl set-mset-minus-replicate-mset}(1) \text{ state-eq}_{NOT}\text{-clauses}$ )  
**ultimately show**  $?case \text{ unfolding } \mu_{CDCL}'\text{-def}$   
**by** ( $\text{metis } (\text{no-types}) T \langle \mu_C' A (\text{remove-cl}_{NOT} C S) = \mu_C' A S \rangle \text{ add-le-cancel-left}$   
 $\mu_C'\text{-def not-le state-eq}_{NOT}\text{-trail}$ )  
**qed**

**lemma**  $\text{cdcl}_{NOT}\text{-clauses-bound}$ :

**assumes**  
 $\text{cdcl}_{NOT} S T$  **and**  
 $\text{inv } S$  **and**  
 $\text{atms-of-mu } (\text{clauses } S) \subseteq A$  **and**  
 $\text{atm-of } (\text{lits-of } (\text{trail } S)) \subseteq A$  **and**  
 $n\text{-d: no-dup } (\text{trail } S)$  **and**  
 $\text{fin-A}[\text{simp}]: \text{finite } A$   
**shows**  $\text{set-mset } (\text{clauses } T) \subseteq \text{set-mset } (\text{clauses } S) \cup \text{build-all-simple-clss } A$   
**using**  $\text{assms}$   
**proof** ( $\text{induction rule: cdcl}_{NOT}\text{-learn-all-induct}$ )  
**case**  $\text{dpll-bj}$   
**then show**  $?case$  **using**  $\text{dpll-bj-clauses}$  **by**  $\text{simp}$   
**next**  
**case**  $\text{forget}_{NOT}$   
**then show**  $?case$  **using**  $\text{clauses-remove-cl}_{NOT}$  **unfolding**  $\text{state-eq}_{NOT}\text{-def}$  **by**  $\text{auto}$   
**next**  
**case** ( $\text{learn } C F K d F' C' L$ ) **note**  $\text{atms-C} = \text{this}(2)$  **and**  $\text{dist} = \text{this}(3)$  **and**  $\text{tauto} = \text{this}(4)$  **and**  
 $T = \text{this}(10)$  **and**  $\text{atms-clss-S} = \text{this}(12)$  **and**  $\text{atms-trail-S} = \text{this}(13)$   
**have**  $\text{atms-of } C \subseteq A$   
**using**  $\text{atms-C atms-clss-S atms-trail-S}$  **by**  $\text{auto}$   
**then have**  $\text{build-all-simple-clss } (\text{atms-of } C) \subseteq \text{build-all-simple-clss } A$   
**by** ( $\text{simp add: build-all-simple-clss-mono}$ )  
**then have**  $C \in \text{build-all-simple-clss } A$   
**using**  $\text{finite dist tauto}$   
**by** ( $\text{auto dest: distinct-mset-not-tautology-implies-in-build-all-simple-clss}$ )  
**then show**  $?case$  **using**  $T n\text{-d}$  **by**  $\text{auto}$   
**qed**

**lemma**  $\text{rtranclp-cdcl}_{NOT}\text{-clauses-bound}$ :

**assumes**  
 $\text{cdcl}_{NOT}^{**} S T$  **and**  
 $\text{inv } S$  **and**  
 $\text{atms-of-mu } (\text{clauses } S) \subseteq A$  **and**  
 $\text{atm-of } (\text{lits-of } (\text{trail } S)) \subseteq A$  **and**  
 $n\text{-d: no-dup } (\text{trail } S)$  **and**  
 $\text{finite: finite } A$   
**shows**  $\text{set-mset } (\text{clauses } T) \subseteq \text{set-mset } (\text{clauses } S) \cup \text{build-all-simple-clss } A$   
**using**  $\text{assms}(1-5)$   
**proof**  $\text{induction}$   
**case**  $\text{base}$   
**then show**  $?case$  **by**  $\text{simp}$   
**next**



**case** (*step*  $T\ U$ ) **note**  $st = \text{this}(1)$  **and**  $cdcl_{NOT} = \text{this}(2)$  **and**  $IH = \text{this}(3)[OF\ \text{this}(4-7)]$  **and**  
 $inv = \text{this}(4)$  **and**  $atms-clss-S = \text{this}(5)$  **and**  $atms-trail-S = \text{this}(6)$  **and**  $finite-clss-S = \text{this}(7)$   
**have**  $inv\ T$   
**using**  $rtrancpl-cdcl_{NOT}-inv\ st\ inv$  **by** *blast*  
**moreover** **have**  $atms-of-mu\ (clauses\ T) \subseteq A$  **and**  $atm-of\ 'lits-of\ (trail\ T) \subseteq A$   
**using**  $rtrancpl-cdcl_{NOT}-trail-clauses-bound[OF\ st]\ inv\ atms-clss-S\ atms-trail-S\ n-d$  **by** *blast+*  
**moreover** **have**  $no-dup\ (trail\ T)$   
**using**  $rtrancpl-cdcl_{NOT}-no-dup[OF\ st\ \langle inv\ S \rangle\ n-d]$  **by** *simp*  
**ultimately** **have**  $set-mset\ (clauses\ U) \subseteq set-mset\ (clauses\ T) \cup build-all-simple-clss\ A$   
**using**  $cdcl_{NOT}\ finite\ n-d$  **by** (*auto simp: cdcl<sub>NOT</sub>-clauses-bound*)  
**then show** *?case* **using**  $IH$  **by** *auto*  
**qed**

**lemma**  $rtrancpl-cdcl_{NOT}-card-clauses-bound$ :

**assumes**  
 $cdcl_{NOT}^{**}\ S\ T$  **and**  
 $inv\ S$  **and**  
 $atms-of-mu\ (clauses\ S) \subseteq A$  **and**  
 $atm-of\ '(lits-of\ (trail\ S)) \subseteq A$  **and**  
 $n-d: no-dup\ (trail\ S)$  **and**  
 $finite: finite\ A$   
**shows**  $card\ (set-mset\ (clauses\ T)) \leq card\ (set-mset\ (clauses\ S)) + 3 \wedge (card\ A)$   
**using**  $rtrancpl-cdcl_{NOT}-clauses-bound[OF\ assms]\ finite$  **by** (*meson Nat.le-trans*  
*build-all-simple-clss-card build-all-simple-clss-finite card-Un-le card-mono finite-UnI*  
*finite-set-mset nat-add-left-cancel-le*)

**lemma**  $rtrancpl-cdcl_{NOT}-card-clauses-bound'$ :

**assumes**  
 $cdcl_{NOT}^{**}\ S\ T$  **and**  
 $inv\ S$  **and**  
 $atms-of-mu\ (clauses\ S) \subseteq A$  **and**  
 $atm-of\ '(lits-of\ (trail\ S)) \subseteq A$  **and**  
 $n-d: no-dup\ (trail\ S)$  **and**  
 $finite: finite\ A$   
**shows**  $card\ \{C \mid C. C \in \# clauses\ T \wedge (tautology\ C \vee \neg distinct-mset\ C)\}$   
 $\leq card\ \{C \mid C. C \in \# clauses\ S \wedge (tautology\ C \vee \neg distinct-mset\ C)\} + 3 \wedge (card\ A)$   
 $(is\ card\ ?T \leq card\ ?S + -)$   
**using**  $rtrancpl-cdcl_{NOT}-clauses-bound[OF\ assms]\ finite$   
**proof** –  
**have**  $?T \subseteq ?S \cup build-all-simple-clss\ A$   
**using**  $rtrancpl-cdcl_{NOT}-clauses-bound[OF\ assms]$  **by** *force*  
**then have**  $card\ ?T \leq card\ (?S \cup build-all-simple-clss\ A)$   
**using**  $finite$  **by** (*simp add: assms(5) build-all-simple-clss-finite card-mono*)  
**then show** *?thesis*  
**by** (*meson le-trans build-all-simple-clss-card card-Un-le local.finite nat-add-left-cancel-le*)  
**qed**

**lemma**  $rtrancpl-cdcl_{NOT}-card-simple-clauses-bound$ :

**assumes**  
 $cdcl_{NOT}^{**}\ S\ T$  **and**  
 $inv\ S$  **and**  
 $atms-of-mu\ (clauses\ S) \subseteq A$  **and**  
 $atm-of\ '(lits-of\ (trail\ S)) \subseteq A$  **and**  
 $n-d: no-dup\ (trail\ S)$  **and**

*finite: finite A*  
**shows**  $\text{card } (\text{set-mset } (\text{clauses } T))$   
 $\leq \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } A)$   
*(is card ?T ≤ card ?S + -)*  
**using**  $\text{rtrancpl-cdcl}_{NOT}\text{-clauses-bound}[OF \text{ assms}] \text{ finite}$   
**proof** –  
**have**  $\bigwedge x. x \in \# \text{ clauses } T \implies \neg \text{tautology } x \implies \text{distinct-mset } x \implies x \in \text{build-all-simple-clss } A$   
**using**  $\text{rtrancpl-cdcl}_{NOT}\text{-clauses-bound}[OF \text{ assms}]$  **by** *(metis (no-types, hide-lams) Un-iff assms(3) atms-of-atms-of-m-mono build-all-simple-clss-mono contra-subsetD distinct-mset-not-tautology-implies-in-build-all-simple-clss local.finite mem-set-mset-iff subset-trans)*  
**then have**  $\text{set-mset } (\text{clauses } T) \subseteq ?S \cup \text{build-all-simple-clss } A$   
**using**  $\text{rtrancpl-cdcl}_{NOT}\text{-clauses-bound}[OF \text{ assms}]$  **by** *auto*  
**then have**  $\text{card}(\text{set-mset } (\text{clauses } T)) \leq \text{card } (?S \cup \text{build-all-simple-clss } A)$   
**using** *finite* **by** *(simp add: assms(5) build-all-simple-clss-finite card-mono)*  
**then show** *?thesis*  
**by** *(meson le-trans build-all-simple-clss-card card-Un-le local.finite nat-add-left-cancel-le)*  
**qed**

**definition**  $\mu_{CDCL}'\text{-bound} :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$  **where**  
 $\mu_{CDCL}'\text{-bound } A \ S =$   
 $((2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))) * (1 + 3 \wedge \text{card } (\text{atms-of-m } A)) * 2$   
 $+ 2 * 3 \wedge (\text{card } (\text{atms-of-m } A))$   
 $+ \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } (\text{atms-of-m } A))$

**lemma**  $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}[simp]:$   
 $\mu_{CDCL}'\text{-bound } A \ (\text{reduce-trail-to}_{NOT} \ M \ S) = \mu_{CDCL}'\text{-bound } A \ S$   
**unfolding**  $\mu_{CDCL}'\text{-bound-def}$  **by** *auto*

**lemma**  $\text{rtrancpl-cdcl}_{NOT}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}:$

**assumes**  
 $\text{cdcl}_{NOT}^{**} \ S \ T$  **and**  
 $\text{inv } S$  **and**  
 $\text{atms-of-mu } (\text{clauses } S) \subseteq \text{atms-of-m } A$  **and**  
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-m } A$  **and**  
 $n\text{-d: no-dup } (\text{trail } S)$  **and**  
 $\text{finite: finite } (\text{atms-of-m } A)$  **and**  
 $U: U \sim \text{reduce-trail-to}_{NOT} \ M \ T$   
**shows**  $\mu_{CDCL}' \ A \ U \leq \mu_{CDCL}'\text{-bound } A \ S$   
**proof** –  
**have**  $((2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))) - \mu_C' \ A \ U$   
 $\leq (2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))$   
**by** *auto*  
**then have**  $((2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))) - \mu_C' \ A \ U$   
 $* (1 + 3 \wedge \text{card } (\text{atms-of-m } A)) * 2$   
 $\leq (2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A)) * (1 + 3 \wedge \text{card } (\text{atms-of-m } A)) * 2$   
**using** *mult-le-mono1* **by** *blast*  
**moreover**  
**have**  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-m } A)) \ T * 2 \leq 2 * 3 \wedge \text{card } (\text{atms-of-m } A)$   
**by** *linarith*  
**moreover have**  $\text{card } (\text{set-mset } (\text{clauses } U))$   
 $\leq \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge \text{card } (\text{atms-of-m } A)$   
**using**  $\text{rtrancpl-cdcl}_{NOT}\text{-card-simple-clauses-bound}[OF \text{ assms}(1-6)] \ U$  **by** *auto*  
**ultimately show** *?thesis*  
**unfolding**  $\mu_{CDCL}'\text{-def}$   $\mu_{CDCL}'\text{-bound-def}$  **by** *linarith*

qed

**lemma** *rtrancpl-cdcl<sub>NOT</sub>- $\mu_{CDCL}$ '-bound*:

**assumes**

*cdcl<sub>NOT</sub>\*\* S T and*

*inv S and*

*atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and*

*atm-of '(lits-of (trail S))  $\subseteq$  atms-of-m A and*

*n-d: no-dup (trail S) and*

*finite: finite (atms-of-m A)*

**shows**  $\mu_{CDCL}' A T \leq \mu_{CDCL}'\text{-bound } A S$

**proof** –

**have**  $\mu_{CDCL}' A (\text{reduce-trail-to}_{NOT} (\text{trail } T) T) = \mu_{CDCL}' A T$

**unfolding**  $\mu_{CDCL}'\text{-def } \mu_C'\text{-def conflicting-bj-clss-def}$  **by** *auto*

**then show** *?thesis using* *rtrancpl-cdcl<sub>NOT</sub>- $\mu_{CDCL}$ '-bound-reduce-trail-to<sub>NOT</sub>[OF assms, of - trail T]*  
*state-eq<sub>NOT</sub>-ref* **by** *fastforce*

qed

**lemma** *rtrancpl- $\mu_{CDCL}$ '-bound-decreasing*:

**assumes**

*cdcl<sub>NOT</sub>\*\* S T and*

*inv S and*

*atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and*

*atm-of '(lits-of (trail S))  $\subseteq$  atms-of-m A and*

*n-d: no-dup (trail S) and*

*finite[simp]: finite (atms-of-m A)*

**shows**  $\mu_{CDCL}'\text{-bound } A T \leq \mu_{CDCL}'\text{-bound } A S$

**proof** –

**have**  $\{C. C \in \# \text{ clauses } T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$

$\subseteq \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$  **(is** *?T  $\subseteq$  ?S)*

**proof** *(rule Set.subsetI)*

**fix** *C* **assume** *C  $\in$  ?T*

**then have** *C-T: C  $\in$  # clauses T and t-d: tautology C  $\vee$   $\neg$  distinct-mset C*  
**by** *auto*

**then have** *C  $\notin$  build-all-simple-clss (atms-of-m A)*

**by** *(auto dest: build-all-simple-clssE)*

**then show** *C  $\in$  ?S*

**using** *C-T* *rtrancpl-cdcl<sub>NOT</sub>-clauses-bound[OF assms]* *t-d* **by** *force*

qed

**then have**  $\text{card } \{C. C \in \# \text{ clauses } T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} \leq$

$\text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$

**by** *(simp add: card-mono)*

**then show** *?thesis*

**unfolding**  $\mu_{CDCL}'\text{-bound-def}$  **by** *auto*

qed

**end** — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt*

## 14.7 CDCL with restarts

### 14.7.1 Definition

**locale** *restart-ops* =

**fixes**

*cdcl<sub>NOT</sub> :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and*

*restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool*

```

begin
inductive cdclNOT-raw-restart :: 'st ⇒ 'st ⇒ bool where
cdclNOT S T ⇒⇒ cdclNOT-raw-restart S T |
restart S T ⇒⇒ cdclNOT-raw-restart S T

end

locale conflict-driven-clause-learning-with-restarts =
conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
propagate-conds inv backjump-conds learn-cond forget-cond
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT remove-clsNOT:: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool and
  backjump-conds :: 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
  learn-cond forget-cond :: 'v clause ⇒ 'st ⇒ bool
begin

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
cdclNOT S T ⇔ restart-ops.cdclNOT-raw-restart cdclNOT (λ- -. False) S T
(is ?C S T ⇔ ?R S T)
proof
  fix S T
  assume ?C S T
  then show ?R S T by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
next
  fix S T
  assume ?R S T
  then show ?C S T
    apply (cases rule: restart-ops.cdclNOT-raw-restart.cases)
    using ⟨?R S T⟩ by fast+
qed

lemma cdclNOT-cdclNOT-raw-restart:
cdclNOT S T ⇒⇒ restart-ops.cdclNOT-raw-restart cdclNOT restart S T
by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
end

```

### 14.7.2 Increasing restarts

To add restarts we need some assumptions on the predicate (called *cdcl*<sub>NOT</sub> 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*<sub>NOT</sub> 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  $cdcl_{NOT}$ -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  $\mu$ -bound taking the same parameter as  $\mu$  and the initial state of the considered  $cdcl_{NOT}$  chain.

```

locale  $cdcl_{NOT}$ -increasing-restarts-ops =
  restart-ops  $cdcl_{NOT}$  restart for
    restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
     $cdcl_{NOT}$  :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool +
fixes
  f :: nat  $\Rightarrow$  nat and
  bound-inv :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  bool and
   $\mu$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat and
   $cdcl_{NOT}$ -inv :: 'st  $\Rightarrow$  bool and
   $\mu$ -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:  $\bigwedge A\ S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$  bound-inv A S  $\Rightarrow$   $cdcl_{NOT}$  S T  $\Rightarrow$  bound-inv A T and
   $cdcl_{NOT}$ -measure:  $\bigwedge A\ S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$  bound-inv A S  $\Rightarrow$   $cdcl_{NOT}$  S T  $\Rightarrow$   $\mu\ A\ T < \mu$ 
A S and
  measure-bound2:  $\bigwedge A\ T\ U. cdcl_{NOT}$ -inv T  $\Rightarrow$  bound-inv A T  $\Rightarrow$   $cdcl_{NOT}^{**}$  T U
     $\Rightarrow \mu\ A\ U \leq \mu$ -bound A T and
  measure-bound4:  $\bigwedge A\ T\ U. cdcl_{NOT}$ -inv T  $\Rightarrow$  bound-inv A T  $\Rightarrow$   $cdcl_{NOT}^{**}$  T U
     $\Rightarrow \mu$ -bound A U  $\leq \mu$ -bound A T and
   $cdcl_{NOT}$ -restart-inv:  $\bigwedge A\ U\ V. cdcl_{NOT}$ -inv U  $\Rightarrow$  restart U V  $\Rightarrow$  bound-inv A U  $\Rightarrow$  bound-inv
A V
and
  exists-bound:  $\bigwedge R\ S. cdcl_{NOT}$ -inv R  $\Rightarrow$  restart R S  $\Rightarrow$   $\exists A. bound$ -inv A S and
   $cdcl_{NOT}$ -inv:  $\bigwedge S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$   $cdcl_{NOT}$  S T  $\Rightarrow$   $cdcl_{NOT}$ -inv T and
   $cdcl_{NOT}$ -inv-restart:  $\bigwedge S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$  restart S T  $\Rightarrow$   $cdcl_{NOT}$ -inv T
begin

```

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

```

assumes
  ( $cdcl_{NOT} \sim n$ ) S T and
   $cdcl_{NOT}$ -inv S
shows  $cdcl_{NOT}$ -inv T
using assms by (induction n arbitrary: T) (auto intro: bound-inv  $cdcl_{NOT}$ -inv)

```

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

```

assumes
  ( $cdcl_{NOT} \sim n$ ) S T and
   $cdcl_{NOT}$ -inv S
  bound-inv A S
shows bound-inv A T
using assms by (induction n arbitrary: T) (auto intro: bound-inv  $cdcl_{NOT}$ - $cdcl_{NOT}$ -inv)

```

**lemma**  $rtrancp$ - $cdcl_{NOT}$ - $cdcl_{NOT}$ -inv:

```

assumes
   $cdcl_{NOT}^{**}$  S T and
   $cdcl_{NOT}$ -inv S

```

**shows**  $cdcl_{NOT-inv} T$   
**using** *assms* **by** *induction* (*auto intro: cdcl<sub>NOT-inv</sub>*)

**lemma** *rtranclp-cdcl<sub>NOT-bound-inv</sub>*:  
**assumes**  
 $cdcl_{NOT}^{**} S T$  **and**  
 $bound-inv A S$  **and**  
 $cdcl_{NOT-inv} S$   
**shows**  $bound-inv A T$   
**using** *assms* **by** *induction* (*auto intro: bound-inv rtranclp-cdcl<sub>NOT-cdcl<sub>NOT-inv</sub></sub>*)

**lemma** *cdcl<sub>NOT-comp-n-le</sub>*:  
**assumes**  
 $(cdcl_{NOT} \sim (Suc\ n)) S T$  **and**  
 $bound-inv A S$   
 $cdcl_{NOT-inv} S$   
**shows**  $\mu A T < \mu A S - n$   
**using** *assms*  
**proof** (*induction n arbitrary: T*)  
**case** 0  
**then show** ?*case* **using** *cdcl<sub>NOT-measure</sub>* **by** *auto*  
**next**  
**case** (*Suc n*) **note**  $IH = this(1)[OF - this(3) this(4)]$  **and**  $S-T = this(2)$  **and**  $b-inv = this(3)$  **and**  
 $c-inv = this(4)$   
**obtain**  $U :: 'st$  **where**  $S-U: (cdcl_{NOT} \sim (Suc\ n)) S U$  **and**  $U-T: cdcl_{NOT} U T$  **using**  $S-T$  **by** *auto*  
**then have**  $\mu A U < \mu A S - n$  **using**  $IH[of\ U]$  **by** *simp*  
**moreover**  
**have**  $bound-inv A U$   
**using**  $S-U\ b-inv\ cdcl_{NOT-bound-inv}\ c-inv$  **by** *blast*  
**then have**  $\mu A T < \mu A U$  **using**  $cdcl_{NOT-measure}[OF - - U-T]\ S-U\ c-inv\ cdcl_{NOT-cdcl_{NOT-inv}}$   
**by** *auto*  
**ultimately show** ?*case* **by** *linarith*  
**qed**

**lemma** *wf-cdcl<sub>NOT</sub>*:  
 $wf \{(T, S). cdcl_{NOT} S T \wedge cdcl_{NOT-inv} S \wedge bound-inv A S\}$  (**is**  $wf\ ?A$ )  
**apply** (*rule wfP-if-measure2[of - -  $\mu A$ ]*)  
**using** *cdcl<sub>NOT-comp-n-le</sub>*[*of 0 - - A*] **by** *auto*

**lemma** *rtranclp-cdcl<sub>NOT-measure</sub>*:  
**assumes**  
 $cdcl_{NOT}^{**} S T$  **and**  
 $bound-inv A S$  **and**  
 $cdcl_{NOT-inv} S$   
**shows**  $\mu A T \leq \mu A S$   
**using** *assms*  
**proof** (*induction rule: rtranclp-induct*)  
**case** *base*  
**then show** ?*case* **by** *auto*  
**next**  
**case** (*step T U*) **note**  $IH = this(3)[OF this(4) this(5)]$  **and**  $st = this(1)$  **and**  $cdcl_{NOT} = this(2)$  **and**  
 $b-inv = this(4)$  **and**  $c-inv = this(5)$   
**have**  $bound-inv A T$   
**by** (*meson cdcl<sub>NOT-bound-inv</sub> rtranclp-imp-relpoup st step.prem*s)  
**moreover have**  $cdcl_{NOT-inv} T$

using  $c\text{-inv}$   $r\text{trancp-cdcl}_{NOT}\text{-cdcl}_{NOT}\text{-inv}$   $st$  by *blast*  
 ultimately have  $\mu A U < \mu A T$  using  $cdcl_{NOT}\text{-measure}[OF - - cdcl_{NOT}]$  by *auto*  
 then show  $?case$  using  $IH$  by *linarith*  
 qed

**lemma**  $cdcl_{NOT}\text{-comp-bounded}$ :

assumes

$bound\text{-inv}$   $A S$  and  $cdcl_{NOT}\text{-inv}$   $S$  and  $m \geq 1 + \mu A S$

shows  $\neg(cdcl_{NOT} \sim^m) S T$

using *assms*  $cdcl_{NOT}\text{-comp-n-le}[of m-1 S T A]$  by *fastforce*

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

**inductive**  $cdcl_{NOT}\text{-restart}$  where

*restart-step*:  $(cdcl_{NOT} \sim^m) S T \implies m \geq f n \implies \text{restart } T U$

$\implies cdcl_{NOT}\text{-restart } (S, n) (U, Suc\ n) \mid$

*restart-full*:  $full1\ cdcl_{NOT} S T \implies cdcl_{NOT}\text{-restart } (S, n) (T, Suc\ n)$

**lemmas**  $cdcl_{NOT}\text{-with-restart-induct} = cdcl_{NOT}\text{-restart.induct}[split\text{-format}(complete),$   
 $OF\ cdcl_{NOT}\text{-increasing-restarts-ops-axioms}]$

**lemma**  $cdcl_{NOT}\text{-restart-cdcl}_{NOT}\text{-raw-restart}$ :

$cdcl_{NOT}\text{-restart } S T \implies cdcl_{NOT}\text{-raw-restart}^{**} (fst\ S) (fst\ T)$

**proof** (*induction rule*:  $cdcl_{NOT}\text{-restart.induct}$ )

**case** (*restart-step*  $m S T n U$ )

**then have**  $cdcl_{NOT}^{**} S T$  by (*meson relpowp-imp-rtrancp*)

**then have**  $cdcl_{NOT}\text{-raw-restart}^{**} S T$  using  $cdcl_{NOT}\text{-raw-restart.intros}(1)$

$r\text{trancp-mono}[of\ cdcl_{NOT}\ cdcl_{NOT}\text{-raw-restart}]$  by *blast*

**moreover have**  $cdcl_{NOT}\text{-raw-restart } T U$

using  $\langle \text{restart } T U \rangle cdcl_{NOT}\text{-raw-restart.intros}(2)$  by *blast*

**ultimately show**  $?case$  by *auto*

**next**

**case** (*restart-full*  $S T$ )

**then have**  $cdcl_{NOT}^{**} S T$  unfolding *full1-def* by *auto*

**then show**  $?case$  using  $cdcl_{NOT}\text{-raw-restart.intros}(1)$

$r\text{trancp-mono}[of\ cdcl_{NOT}\ cdcl_{NOT}\text{-raw-restart}]$  by *auto*

qed

**lemma**  $cdcl_{NOT}\text{-with-restart-bound-inv}$ :

assumes

$cdcl_{NOT}\text{-restart } S T$  and

$bound\text{-inv}$   $A (fst\ S)$  and

$cdcl_{NOT}\text{-inv} (fst\ S)$

shows  $bound\text{-inv}$   $A (fst\ T)$

using *assms* **apply** (*induction rule*:  $cdcl_{NOT}\text{-restart.induct}$ )

**prefer** 2 **apply** (*metis*  $r\text{trancp-unfold}$  *fstI* *full1-def*  $r\text{trancp-cdcl}_{NOT}\text{-bound-inv}$ )

by (*metis*  $cdcl_{NOT}\text{-bound-inv}$   $cdcl_{NOT}\text{-cdcl}_{NOT}\text{-inv}$   $cdcl_{NOT}\text{-restart-inv}$  *fst-conv*)

**lemma**  $cdcl_{NOT}\text{-with-restart-cdcl}_{NOT}\text{-inv}$ :

assumes

$cdcl_{NOT}\text{-restart } S T$  and

$cdcl_{NOT}\text{-inv} (fst\ S)$

shows  $cdcl_{NOT}\text{-inv} (fst\ T)$

using *assms* **apply** *induction*

**apply** (*metis*  $cdcl_{NOT}\text{-cdcl}_{NOT}\text{-inv}$   $cdcl_{NOT}\text{-inv-restart}$  *fst-conv*)

**apply** (*metis fstI full-def full-unfold rtranclp-cdcl<sub>NOT</sub>-cdcl<sub>NOT</sub>-inv*)  
**done**

**lemma** *rtranclp-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)*  
**using** *assms* **by** *induction (auto intro: cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv)*

**lemma** *rtranclp-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)*  
**using** *assms* **apply** *induction*  
**apply** (*simp add: cdcl<sub>NOT</sub>-cdcl<sub>NOT</sub>-inv cdcl<sub>NOT</sub>-with-restart-bound-inv*)  
**using** *cdcl<sub>NOT</sub>-with-restart-bound-inv rtranclp-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv* **by** *blast*

**lemma** *cdcl<sub>NOT</sub>-with-restart-increasing-number*:

*cdcl<sub>NOT</sub>-restart S T  $\implies$  snd T = 1 + snd S*  
**by** (*induction rule: cdcl<sub>NOT</sub>-restart.induct*) *auto*  
**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*  
**for**

*trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and*  
*clauses :: 'st  $\Rightarrow$  'v clauses and*  
*prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and*  
*tl-trail :: 'st  $\Rightarrow$  'st and*  
*add-cls<sub>NOT</sub> remove-cls<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<sub>NOT</sub>-inv :: 'st  $\Rightarrow$  bool and*  
 *$\mu$ -bound :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat +*

**assumes**

*measure-bound:  $\bigwedge A T V n. cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A T$*   
 $\implies cdcl_{NOT}\text{-restart } (T, n) (V, Suc\ n) \implies \mu\ A\ V \leq \mu\text{-bound } A\ T$  **and**  
*cdcl<sub>NOT</sub>-raw-restart- $\mu$ -bound:*  
*cdcl<sub>NOT</sub>-restart (T, a) (V, b)  $\implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A\ T$*   
 $\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$

**begin**

**lemma** *rtranclp-cdcl<sub>NOT</sub>-raw-restart- $\mu$ -bound*:

*cdcl<sub>NOT</sub>-restart\*\* (T, a) (V, b)  $\implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A\ T$*   
 $\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$   
**apply** (*induction rule: rtranclp-induct2*)  
**apply** *simp*  
**by** (*metis cdcl<sub>NOT</sub>-raw-restart- $\mu$ -bound dual-order.trans fst-conv*  
*rtranclp-cdcl<sub>NOT</sub>-with-restart-bound-inv rtranclp-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv*)



```

lemma cdclNOT-raw-restart-measure-bound:
  cdclNOT-restart (T, a) (V, b)  $\implies$  cdclNOT-inv T  $\implies$  bound-inv A T
   $\implies \mu$  A V  $\leq$   $\mu$ -bound A T
apply (cases rule: cdclNOT-restart.cases)
  apply simp
  using measure-bound relpoup-imp-rtrancp apply fastforce
by (metis full-def full-unfold measure-bound2 prod.inject)

lemma rtrancp-cdclNOT-raw-restart-measure-bound:
  cdclNOT-restart** (T, a) (V, b)  $\implies$  cdclNOT-inv T  $\implies$  bound-inv A T
   $\implies \mu$  A V  $\leq$   $\mu$ -bound A T
apply (induction rule: rtrancp-induct2)
apply (simp add: measure-bound2)
by (metis dual-order.trans fst-conv measure-bound2 r-into-rtrancp rtrancp.rtrancp-refl
  rtrancp-cdclNOT-with-restart-bound-inv rtrancp-cdclNOT-with-restart-cdclNOT-inv
  rtrancp-cdclNOT-raw-restart- $\mu$ -bound)

lemma wf-cdclNOT-restart:
  wf {(T, S). cdclNOT-restart S T  $\wedge$  cdclNOT-inv (fst S)} (is wf ?A)
proof (rule ccontr)
assume  $\neg$  ?thesis
then obtain g where
  g:  $\bigwedge i.$  cdclNOT-restart (g i) (g (Suc i)) and
  cdclNOT-inv-g:  $\bigwedge i.$  cdclNOT-inv (fst (g i))
unfolding wf-iff-no-infinite-down-chain by fast

have snd-g:  $\bigwedge i.$  snd (g i) = i + snd (g 0)
apply (induct-tac i)
apply simp
by (metis Suc-eq-plus1-left add commute add.left-commute
  cdclNOT-with-restart-increasing-number g)
then have snd-g-0:  $\bigwedge i.$  i > 0  $\implies$  snd (g i) = i + snd (g 0)
by blast
have unbounded-f-g: unbounded ( $\lambda i.$  f (snd (g i)))
using f unfolding bounded-def by (metis add commute f less-or-eq-imp-le snd-g
  not-bounded-nat-exists-larger not-le ordered-cancel-comm-monoid-diff-class.le-iff-add)

{ fix i
  have H:  $\bigwedge T$  Ta m. (cdclNOT  $\rightsquigarrow$  m) T Ta  $\implies$  no-step cdclNOT T  $\implies$  m = 0
  apply (case-tac m) apply simp by (meson relpoup-E2)
  have  $\exists$  T m. (cdclNOT  $\rightsquigarrow$  m) (fst (g i)) T  $\wedge$  m  $\geq$  f (snd (g i))
  using g[of i] apply (cases rule: cdclNOT-restart.cases)
  apply auto[]
  using g[of Suc i] f-ge-1 apply (cases rule: cdclNOT-restart.cases)
  apply (auto simp add: full1-def full-def dest: H dest: trancpD)
  using H Suc-leI leD by blast
} note H = this
obtain A where bound-inv A (fst (g 1))
using g[of 0] cdclNOT-inv-g[of 0] apply (cases rule: cdclNOT-restart.cases)
apply (metis One-nat-def cdclNOT-inv exists-bound fst-conv relpoup-imp-rtrancp
  rtrancp-induct)
using H[of 1] unfolding full1-def by (metis One-nat-def Suc-eq-plus1 diff-is-0-eq' diff-zero
  f-ge-1 fst-conv le-add2 relpoup-E2 snd-conv)
let ?j =  $\mu$ -bound A (fst (g 1)) + 1

```

```

obtain  $j$  where
   $j: f \text{ (snd } (g \ j)) > ?j$  and  $j > 1$ 
  using unbounded-f-g not-bounded-nat-exists-larger by blast
{
  fix  $i \ j$ 
  have  $cdcl_{NOT}\text{-with-restart}: j \geq i \implies cdcl_{NOT}\text{-restart}^{**} (g \ i) (g \ j)$ 
    apply (induction j)
    apply simp
    by (metis g le-Suc-eq rtrancp.rtrancl-into-rtrancl rtrancp.rtrancl-refl)
} note  $cdcl_{NOT}\text{-restart} = \text{this}$ 
have  $cdcl_{NOT}\text{-inv} (fst (g \ (Suc \ 0)))$ 
  by (simp add: cdcl_{NOT}\text{-inv-g})
have  $cdcl_{NOT}\text{-restart}^{**} (fst (g \ 1), snd (g \ 1)) (fst (g \ j), snd (g \ j))$ 
  using  $\langle j > 1 \rangle$  by (simp add: cdcl_{NOT}\text{-restart})
have  $\mu \ A \ (fst (g \ j)) \leq \mu\text{-bound } A \ (fst (g \ 1))$ 
  apply (rule rtrancp-cdcl_{NOT}\text{-raw-restart-measure-bound})
  using  $\langle cdcl_{NOT}\text{-restart}^{**} (fst (g \ 1), snd (g \ 1)) (fst (g \ j), snd (g \ j)) \rangle$  apply blast
  apply (simp add: cdcl_{NOT}\text{-inv-g})
  using  $\langle \text{bound-inv } A \ (fst (g \ 1)) \rangle$  apply simp
done
then have  $\mu \ A \ (fst (g \ j)) \leq ?j$ 
  by auto
have  $\text{inv: bound-inv } A \ (fst (g \ j))$ 
  using  $\langle \text{bound-inv } A \ (fst (g \ 1)) \rangle \langle cdcl_{NOT}\text{-inv} (fst (g \ (Suc \ 0))) \rangle$ 
   $\langle cdcl_{NOT}\text{-restart}^{**} (fst (g \ 1), snd (g \ 1)) (fst (g \ j), snd (g \ j)) \rangle$ 
  rtrancp-cdcl_{NOT}\text{-with-restart-bound-inv} by auto
obtain  $T \ m$  where
   $cdcl_{NOT}\text{-m}: (cdcl_{NOT} \rightsquigarrow m) (fst (g \ j)) \ T$  and
   $f\text{-m}: f \text{ (snd } (g \ j)) \leq m$ 
  using  $H[\text{of } j]$  by blast
have  $?j < m$ 
  using  $f\text{-m } j \text{ Nat.le-trans}$  by linarith

then show False
  using  $\langle \mu \ A \ (fst (g \ j)) \leq \mu\text{-bound } A \ (fst (g \ 1)) \rangle$ 
   $cdcl_{NOT}\text{-comp-bounded}[OF \ \text{inv } cdcl_{NOT}\text{-inv-g}, \text{ of } ] \ cdcl_{NOT}\text{-inv-g } cdcl_{NOT}\text{-m}$ 
   $\langle ?j < m \rangle$  by auto
qed

lemma  $cdcl_{NOT}\text{-restart-steps-bigger-than-bound}$ :
assumes
   $cdcl_{NOT}\text{-restart } S \ T$  and
   $\text{bound-inv } A \ (fst \ S)$  and
   $cdcl_{NOT}\text{-inv} (fst \ S)$  and
   $f \text{ (snd } S) > \mu\text{-bound } A \ (fst \ S)$ 
shows  $\text{full1 } cdcl_{NOT} \ (fst \ S) \ (fst \ T)$ 
using assms
proof (induction rule: cdcl_{NOT}\text{-restart.induct})
case restart-full
then show  $?case$  by auto
next
case (restart-step m S T n U) note  $st = \text{this}(1)$  and  $f = \text{this}(2)$  and  $\text{bound-inv} = \text{this}(4)$  and
   $cdcl_{NOT}\text{-inv} = \text{this}(5)$  and  $\mu = \text{this}(6)$ 
then obtain  $m'$  where  $m: m = Suc \ m'$  by (cases m) auto
have  $\mu \ A \ S - m' = 0$ 

```

using  $f$  bound-inv cdcl<sub>NOT</sub>-inv  $\mu$   $m$  rtrancpl-cdcl<sub>NOT</sub>-raw-restart-measure-bound by fastforce  
 then have False using cdcl<sub>NOT</sub>-comp-n-le[of  $m'$   $S$   $T$   $A$ ] restart-step unfolding  $m$  by simp  
 then show ?case by fast  
 qed

**lemma** rtrancpl-cdcl<sub>NOT</sub>-with-inv-inv-rtrancpl-cdcl<sub>NOT</sub>:

assumes

inv: cdcl<sub>NOT</sub>-inv  $S$  and

binv: bound-inv  $A$   $S$

shows  $(\lambda S T. \text{cdcl}_{\text{NOT}} S T \wedge \text{cdcl}_{\text{NOT}}\text{-inv } S \wedge \text{bound-inv } A S)^{**} S T \longleftrightarrow \text{cdcl}_{\text{NOT}}^{**} S T$   
 (is  $?A^{**} S T \longleftrightarrow ?B^{**} S T$ )

apply (rule iffI)

using rtrancpl-mono[of  $?A$   $?B$ ] apply blast

apply (induction rule: rtrancpl-induct)

using inv binv apply simp

by (metis (mono-tags, lifting) binv inv rtrancpl.simps rtrancpl-cdcl<sub>NOT</sub>-bound-inv  
 rtrancpl-cdcl<sub>NOT</sub>-cdcl<sub>NOT</sub>-inv)

**lemma** no-step-cdcl<sub>NOT</sub>-restart-no-step-cdcl<sub>NOT</sub>:

assumes

$n$ -s: no-step cdcl<sub>NOT</sub>-restart  $S$  and

inv: cdcl<sub>NOT</sub>-inv (fst  $S$ ) and

binv: bound-inv  $A$  (fst  $S$ )

shows no-step cdcl<sub>NOT</sub> (fst  $S$ )

**proof** (rule ccontr)

assume  $\neg$  ?thesis

then obtain  $T$  where  $T$ : cdcl<sub>NOT</sub> (fst  $S$ )  $T$

by blast

then obtain  $U$  where  $U$ : full  $(\lambda S T. \text{cdcl}_{\text{NOT}} S T \wedge \text{cdcl}_{\text{NOT}}\text{-inv } S \wedge \text{bound-inv } A S) T U$

using wf-exists-normal-form-full[OF wf-cdcl<sub>NOT</sub>, of  $A$   $T$ ] by auto

moreover have inv- $T$ : cdcl<sub>NOT</sub>-inv  $T$

using  $\langle \text{cdcl}_{\text{NOT}} (\text{fst } S) T \rangle$  cdcl<sub>NOT</sub>-inv inv by blast

moreover have b-inv- $T$ : bound-inv  $A$   $T$

using  $\langle \text{cdcl}_{\text{NOT}} (\text{fst } S) T \rangle$  binv bound-inv inv by blast

ultimately have full cdcl<sub>NOT</sub>  $T$   $U$

using rtrancpl-cdcl<sub>NOT</sub>-with-inv-inv-rtrancpl-cdcl<sub>NOT</sub> rtrancpl-cdcl<sub>NOT</sub>-bound-inv  
 rtrancpl-cdcl<sub>NOT</sub>-cdcl<sub>NOT</sub>-inv unfolding full-def by blast

then have full1 cdcl<sub>NOT</sub> (fst  $S$ )  $U$

using  $T$  full-full1 by metis

then show False by (metis  $n$ -s prod.collapse restart-full)

qed

end

## 14.8 Merging backjump and learning

**locale** cdcl<sub>NOT</sub>-merge-bj-learn-ops =

dpll-state trail clauses prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> +

decide-ops trail clauses prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> +

forget-ops trail clauses prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> forget-cond +

propagate-ops trail clauses prepend-trail tl-trail add-cl<sub>NOT</sub> remove-cl<sub>NOT</sub> propagate-conds

**for**

trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and

clauses :: 'st  $\Rightarrow$  'v clauses and

prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and

tl-trail :: 'st  $\Rightarrow$  'st and

```

    add-clNOT remove-clNOT:: 'v clause ⇒ 'st ⇒ 'st and
    propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
    forget-cond :: 'v clause ⇒ 'st ⇒ bool +
fixes backjump-l-cond :: 'v clause ⇒ 'v literal ⇒ 'st ⇒ bool
begin
inductive backjump-l where
backjump-l: trail S = F' @ Marked K () # F
    ⇒ no-dup (trail S)
    ⇒ T ~ prepend-trail (Propagated L ()) (reduce-trail-toNOT F (add-clNOT (C' + {#L#}) S))
    ⇒ C ∈ # clauses S
    ⇒ trail S ⊨as CNot C
    ⇒ undefined-lit F L
    ⇒ atm-of L ∈ atms-of-mu (clauses S) ∪ atm-of ' (lits-of (trail S))
    ⇒ clauses S ⊨pm C' + {#L#}
    ⇒ F ⊨as CNot C'
    ⇒ backjump-l-cond C' L T
    ⇒ backjump-l S T
inductive-cases backjump-lE: backjump-l S T

inductive cdclNOT-merged-bj-learn :: 'st ⇒ 'st ⇒ bool for S :: 'st where
cdclNOT-merged-bj-learn-decideNOT: decideNOT S S' ⇒ cdclNOT-merged-bj-learn S S' |
cdclNOT-merged-bj-learn-propagateNOT: propagateNOT S S' ⇒ cdclNOT-merged-bj-learn S S' |
cdclNOT-merged-bj-learn-backjump-l: backjump-l S S' ⇒ cdclNOT-merged-bj-learn S S' |
cdclNOT-merged-bj-learn-forgetNOT: forgetNOT S S' ⇒ cdclNOT-merged-bj-learn S S'

lemma cdclNOT-merged-bj-learn-no-dup-inv:
cdclNOT-merged-bj-learn S T ⇒ no-dup (trail S) ⇒ no-dup (trail T)
apply (induction rule: cdclNOT-merged-bj-learn.induct)
    using defined-lit-map apply fastforce
    using defined-lit-map apply fastforce
apply (auto simp: defined-lit-map elim!: backjump-lE)[]
using forgetNOT.simps apply auto[1]
done
end

locale cdclNOT-merge-bj-learn-proxy =
cdclNOT-merge-bj-learn-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
propagate-conds forget-conds λC L S. backjump-l-cond C L S ∧ distinct-mset (C + {#L#})
∧ ¬tautology (C + {#L#})
for
trail :: 'st ⇒ ('v, unit, unit) marked-lits and
clauses :: 'st ⇒ 'v clauses and
prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
tl-trail :: 'st ⇒ 'st and
add-clNOT remove-clNOT:: 'v clause ⇒ 'st ⇒ 'st and
propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
forget-conds :: 'v clause ⇒ 'st ⇒ bool and
backjump-l-cond :: 'v clause ⇒ 'v literal ⇒ 'st ⇒ bool +
fixes
inv :: 'st ⇒ bool
assumes
bj-can-jump:
    ∧ S C F' K F L.
    inv S
    ⇒ trail S = F' @ Marked K () # F

```

$\Rightarrow C \in \# \text{ clauses } S$   
 $\Rightarrow \text{trail } S \models_{as} C \text{Not } C$   
 $\Rightarrow \text{undefined-lit } F \ L$   
 $\Rightarrow \text{atm-of } L \in \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (F' @ \text{Marked } K \ () \ # \ F))$   
 $\Rightarrow \text{clauses } S \models_{pm} C' + \{\#L\#\}$   
 $\Rightarrow F \models_{as} C \text{Not } C'$   
 $\Rightarrow \neg \text{no-step backjump-l } S \text{ and}$   
 $\text{cdcl-merged-inv: } \bigwedge S \ T. \text{ cdcl}_{NOT}\text{-merged-bj-learn } S \ T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$

**begin**

**abbreviation** *backjump-conds* **where**

*backjump-conds*  $\equiv \lambda C \ L \ -. \ \text{distinct-mset } (C + \{\#L\#\}) \wedge \neg \text{tautology } (C + \{\#L\#\})$

**sublocale** *dp11-with-backjumping-ops* *trail clauses prepend-trail tl-trail add-cls<sub>NOT</sub> remove-cls<sub>NOT</sub>*  
*propagate-conds inv backjump-conds*

**proof** (*unfold-locales, goal-cases*)

**case** 1

**{ fix** *S S'*

**assume** *bj: backjump-l S S' and no-dup (trail S)*

**then obtain** *F' K F L C' C where*

*S': S' ~ prepend-trail (Propagated L ()) (reduce-trail-to<sub>NOT</sub> F*  
*(tl-trail(add-cls<sub>NOT</sub> (C' + {\#L\#}) S)))*

**and**

*tr-S: trail S = F' @ Marked K () # F and*

*C: C ∈ # clauses S and*

*tr-S-C: trail S ⊨<sub>as</sub> CNot C and*

*undef-L: undefined-lit F L and*

*atm-L: atm-of L ∈ atms-of-mu (clauses S) ∪ atm-of ' lits-of (trail S) and*

*cls-S-C': clauses S ⊨<sub>pm</sub> C' + {\#L\#} and*

*F-C': F ⊨<sub>as</sub> CNot C' and*

*dist: distinct-mset (C' + {\#L\#}) and*

*not-tauto: ¬ tautology (C' + {\#L\#})*

**by** (*elim backjump-lE*) *simp*

**have**  $\exists S'. \text{backjumping-ops.backjump trail clauses prepend-trail tl-trail backjump-conds } S \ S'$

**apply** *rule*

**apply** (*rule backjumping-ops.backjump.intros*)

**apply** *unfold-locales*

**using** *tr-S* **apply** *simp*

**apply** (*rule state-eq<sub>NOT</sub>-ref*)

**using** *C* **apply** *simp*

**using** *tr-S-C* **apply** *simp*

**using** *undef-L* **apply** *simp*

**using** *atm-L* **apply** *simp*

**using** *cls-S-C'* **apply** *simp*

**using** *F-C'* **apply** *simp*

**using** *dist not-tauto* **apply** *simp*

**done**

**} note** *H = this(1)*

**then show** *?case using 1 bj-can-jump by meson*

**qed**

**end**

**locale** *cdcl<sub>NOT</sub>-merge-bj-learn-proxy2* =

*cdcl<sub>NOT</sub>-merge-bj-learn-proxy trail clauses prepend-trail tl-trail add-cls<sub>NOT</sub> remove-cls<sub>NOT</sub>*

```

    propagate-conds forget-conds backjump-l-cond inv
  for
    trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
    clauses :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
    tl-trail :: 'st  $\Rightarrow$  'st and
    add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
    inv :: 'st  $\Rightarrow$  bool and
    forget-conds :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool and
    backjump-l-cond :: 'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  bool
  begin

  sublocale conflict-driven-clause-learning-ops trail clauses prepend-trail tl-trail add-clNOT
    remove-clNOT propagate-conds inv backjump-conds  $\lambda C$  -. distinct-mset  $C \wedge \neg$ tautology  $C$ 
    forget-conds
  by unfold-locales
end

locale cdclNOT-merge-bj-learn =
  cdclNOT-merge-bj-learn-proxy2 trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
    propagate-conds inv forget-conds backjump-l-cond
  for
    trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
    clauses :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
    tl-trail :: 'st  $\Rightarrow$  'st and
    add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
    inv :: 'st  $\Rightarrow$  bool and
    forget-conds :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool and
    backjump-l-cond :: 'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  bool +
  assumes
    dpll-bj-inv:  $\bigwedge S T. \text{dpll-bj } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$  and
    learn-inv:  $\bigwedge S T. \text{learn } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 
  begin

  interpretation cdclNOT:
    conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
    propagate-conds inv backjump-conds  $\lambda C$  -. distinct-mset  $C \wedge \neg$ tautology  $C$  forget-conds
  apply unfold-locales
  apply (simp only: cdclNOT.simps)
  using cdclNOT-merged-bj-learn-forgetNOT cdcl-merged-inv learn-inv
  by (auto simp add: cdclNOT.simps dpll-bj-inv)

  lemma backjump-l-learn-backjump:
    assumes bt: backjump-l  $S T$  and inv: inv  $S$  and n-d: no-dup (trail  $S$ )
    shows  $\exists C' L. \text{learn } S (\text{add-cl}_{\text{NOT}} (C' + \{\#L\}) S)$ 
       $\wedge \text{backjump } (\text{add-cl}_{\text{NOT}} (C' + \{\#L\}) S) T$ 
       $\wedge \text{atms-of } (C' + \{\#L\}) \subseteq \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$ 
  proof -
    obtain  $C F' K F L l C'$  where
      tr-S: trail  $S = F' @ \text{Marked } K () \# F$  and
      T:  $T \sim \text{prepend-trail } (\text{Propagated } L l) (\text{reduce-trail-to}_{\text{NOT}} F (\text{add-cl}_{\text{NOT}} (C' + \{\#L\}) S))$  and
      C-clS:  $C \in \# \text{clauses } S$  and

```

*tr-S-CNot-C*:  $\text{trail } S \models_{\text{as}} \text{CNot } C$  **and**  
*undef*: *undefined-lit*  $F \ L$  **and**  
*atm-L*:  $\text{atm-of } L \in \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$  **and**  
*clss-C*:  $\text{clauses } S \models_{\text{pm}} C' + \{\#L\# \}$  **and**  
 $F \models_{\text{as}} \text{CNot } C'$  **and**  
*distinct*:  $\text{distinct-mset } (C' + \{\#L\# \})$  **and**  
*not-tauto*:  $\neg \text{tautology } (C' + \{\#L\# \})$   
**using** *bt inv* **by** (*force elim!*: *backjump-lE*)  
**have** *atms-C'*:  $\text{atms-of } C' \subseteq \text{atm-of } ' (\text{lits-of } F)$   
**proof** –  
**obtain** *ll* ::  $'v \Rightarrow ('v \text{ literal} \Rightarrow 'v) \Rightarrow 'v \text{ literal set} \Rightarrow 'v \text{ literal}$  **where**  
 $\forall v f L. v \notin f \text{ ' } L \vee v = f \ (ll \ v \ f \ L) \wedge ll \ v \ f \ L \in L$   
**by** *moura*  
**then show** *?thesis unfolding tr-S*  
**by** (*metis* (*no-types*)  $\langle F \models_{\text{as}} \text{CNot } C' \rangle$  *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*  
*atms-of-def in-CNot-implies-uminus(2)* *mem-set-mset-iff subsetI*)  
**qed**  
**then have** *atms-of*  $(C' + \{\#L\# \}) \subseteq \text{atms-of-mu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$   
**using** *atm-L tr-S* **by** *auto*  
**moreover have** *learn*:  $\text{learn } S \ (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) \ S)$   
**apply** (*rule learn.intros*)  
**apply** (*rule clss-C*)  
**using** *atms-C' atm-L* **apply** (*fastforce simp add: tr-S in-plus-implies-atm-of-on-atms-of-m*)[]  
**apply** *standard*  
**apply** (*rule distinct*)  
**apply** (*rule not-tauto*)  
**apply** *simp*  
**done**  
**moreover have** *bj*:  $\text{backjump } (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) \ S) \ T$   
**apply** (*rule backjump.intros*)  
**using**  $\langle F \models_{\text{as}} \text{CNot } C' \rangle$  *C-cl-S tr-S-CNot-C undef T distinct not-tauto n-d*  
**by** (*auto simp: tr-S state-eq<sub>NOT</sub>-def simp del: state-simp<sub>NOT</sub>*)  
**ultimately show** *?thesis* **by** *auto*  
**qed**

**lemma** *cdcl<sub>NOT</sub>-merged-bj-learn-is-tranclp-cdcl<sub>NOT</sub>*:  
 $\text{cdcl}_{\text{NOT}}\text{-merged-bj-learn } S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{no-dup } (\text{trail } S) \Longrightarrow \text{cdcl}_{\text{NOT}}^{++} \ S \ T$   
**proof** (*induction rule: cdcl<sub>NOT</sub>-merged-bj-learn.induct*)  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-decide<sub>NOT</sub> T*)  
**then have**  $\text{cdcl}_{\text{NOT}} \ S \ T$   
**using** *bj-decide<sub>NOT</sub> cdcl<sub>NOT</sub>.simps* **by** *fastforce*  
**then show** *?case* **by** *auto*  
**next**  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-propagate<sub>NOT</sub> T*)  
**then have**  $\text{cdcl}_{\text{NOT}} \ S \ T$   
**using** *bj-propagate<sub>NOT</sub> cdcl<sub>NOT</sub>.simps* **by** *fastforce*  
**then show** *?case* **by** *auto*  
**next**  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-forget<sub>NOT</sub> T*)  
**then have**  $\text{cdcl}_{\text{NOT}} \ S \ T$   
**using** *c-forget<sub>NOT</sub>* **by** *blast*  
**then show** *?case* **by** *auto*  
**next**  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-backjump-l T*) **note** *bt = this(1)* **and** *inv = this(2)* **and**  
*n-d = this(3)*

**show** ?case  
**using** backjump-l-learn-backjump[OF bt inv] n-d  
**by** (metis (no-types, lifting) bj-backjump c-dpll-bj c-learn  
 tranclp.r-into-trancl tranclp.trancl-into-trancl)  
**qed**

**lemma** rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtranclp-cdcl<sub>NOT</sub>-and-inv:

cdcl<sub>NOT</sub>-merged-bj-learn\*\* S T  $\implies$  inv S  $\implies$  no-dup (trail S)  $\implies$  cdcl<sub>NOT</sub>\*\* S T  $\wedge$  inv T

**proof** (induction rule: rtranclp-induct)

**case** base

**then show** ?case **by** auto

**next**

**case** (step T U) **note** st = this(1) **and** cdcl<sub>NOT</sub> = this(2) **and** IH = this(3)[OF this(4-)] **and**  
 inv = this(4) **and** n-d = this(5)

**have** cdcl<sub>NOT</sub>\*\* T U

**using** cdcl<sub>NOT</sub>-merged-bj-learn-is-tranclp-cdcl<sub>NOT</sub>[OF cdcl<sub>NOT</sub>] IH

cdcl<sub>NOT</sub>.rtranclp-cdcl<sub>NOT</sub>-no-dup inv n-d **by** auto

**then have** cdcl<sub>NOT</sub>\*\* S U **using** IH **by** fastforce

**moreover have** inv U **using** n-d IH  $\langle$ cdcl<sub>NOT</sub>\*\* T U $\rangle$  cdcl<sub>NOT</sub>.rtranclp-cdcl<sub>NOT</sub>-inv **by** blast

**ultimately show** ?case **using** st **by** fast

**qed**

**lemma** rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtranclp-cdcl<sub>NOT</sub>:

cdcl<sub>NOT</sub>-merged-bj-learn\*\* S T  $\implies$  inv S  $\implies$  no-dup (trail S)  $\implies$  cdcl<sub>NOT</sub>\*\* S T

**using** rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtranclp-cdcl<sub>NOT</sub>-and-inv **by** blast

**lemma** rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-inv:

cdcl<sub>NOT</sub>-merged-bj-learn\*\* S T  $\implies$  inv S  $\implies$  no-dup (trail S)  $\implies$  inv T

**using** rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtranclp-cdcl<sub>NOT</sub>-and-inv **by** blast

**definition**  $\mu_C' :: 'v$  literal multiset set  $\Rightarrow$  'st  $\Rightarrow$  nat **where**

$\mu_C' A T \equiv \mu_C (1 + \text{card} (\text{atms-of-m } A)) (2 + \text{card} (\text{atms-of-m } A)) (\text{trail-weight } T)$

**definition**  $\mu_{CDCL}'$ -merged :: 'v literal multiset set  $\Rightarrow$  'st  $\Rightarrow$  nat **where**

$\mu_{CDCL}'$ -merged A T  $\equiv$

$((2 + \text{card} (\text{atms-of-m } A)) \wedge (1 + \text{card} (\text{atms-of-m } A)) - \mu_C' A T) * 2 + \text{card} (\text{set-mset} (\text{clauses } T))$

**lemma** cdcl<sub>NOT</sub>-decreasing-measure':

**assumes**

cdcl<sub>NOT</sub>-merged-bj-learn S T **and**

inv: inv S **and**

atm-clss: atm-of-mu (clauses S)  $\subseteq$  atm-of-m A **and**

atm-trail: atm-of ' lits-of (trail S)  $\subseteq$  atm-of-m A **and**

n-d: no-dup (trail S) **and**

fin-A: finite A

**shows**  $\mu_{CDCL}'$ -merged A T <  $\mu_{CDCL}'$ -merged A S

**using** assms(1)

**proof** induction

**case** (cdcl<sub>NOT</sub>-merged-bj-learn-decide<sub>NOT</sub> T)

**have** clauses S = clauses T

**using** cdcl<sub>NOT</sub>-merged-bj-learn-decide<sub>NOT</sub>.hypos **by** auto

**moreover have**

$(2 + \text{card} (\text{atms-of-m } A)) \wedge (1 + \text{card} (\text{atms-of-m } A))$   
 $- \mu_C (1 + \text{card} (\text{atms-of-m } A)) (2 + \text{card} (\text{atms-of-m } A)) (\text{trail-weight } T)$   
 $< (2 + \text{card} (\text{atms-of-m } A)) \wedge (1 + \text{card} (\text{atms-of-m } A))$



$\mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight } S)$   
**apply** (*rule dpll-bj-trail-mes-decreasing-prop*)  
**using** *cdcl<sub>NOT</sub>-merged-bj-learn-decide<sub>NOT</sub> fin-A atm-clss atm-trail n-d inv*  
**by** (*simp-all add: bj-decide<sub>NOT</sub> cdcl<sub>NOT</sub>-merged-bj-learn-decide<sub>NOT</sub>.hyps*)  
**ultimately show** ?case  
**unfolding**  $\mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def}$  **by** *simp*  
**next**  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-propagate<sub>NOT</sub> T*)  
**have** *clauses S = clauses T*  
**using** *cdcl<sub>NOT</sub>-merged-bj-learn-propagate<sub>NOT</sub>.hyps*  
**by** (*simp add: bj-propagate<sub>NOT</sub> inv dpll-bj-clauses*)  
**moreover have**  
 $(2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight } T)$   
 $< (2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight } S)$   
**apply** (*rule dpll-bj-trail-mes-decreasing-prop*)  
**using** *inv n-d atm-clss atm-trail fin-A* **by** (*simp-all add: bj-propagate<sub>NOT</sub>*  
*cdcl<sub>NOT</sub>-merged-bj-learn-propagate<sub>NOT</sub>.hyps*)  
**ultimately show** ?case  
**unfolding**  $\mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def}$  **by** *simp*  
**next**  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-forget<sub>NOT</sub> T*)  
**have** *card (set-mset (clauses T)) < card (set-mset (clauses S))*  
**using**  $\langle \text{forget}_{NOT} S T \rangle$  **by** (*metis card-Diff1-less*  
*cdcl<sub>NOT</sub>-merged-bj-learn-forget<sub>NOT</sub>.hyps clauses-remove-cls<sub>NOT</sub> finite-set-mset forgetE*  
*mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1) state-eq<sub>NOT</sub>-clauses*)  
**moreover**  
**have** *trail S = trail T*  
**using**  $\langle \text{forget}_{NOT} S T \rangle$  **by** (*auto elim: forgetE*)  
**then have**  
 $(2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight } T)$   
 $= (2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight } S)$   
**by** *auto*  
**ultimately show** ?case  
**unfolding**  $\mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def}$  **by** *simp*  
**next**  
**case** (*cdcl<sub>NOT</sub>-merged-bj-learn-backjump-l T*) **note** *bj-l = this(1)*  
**obtain** *C' L* **where**  
*learn: learn S (add-cls<sub>NOT</sub> (C' + {#L#}) S) and*  
*bj: backjump (add-cls<sub>NOT</sub> (C' + {#L#}) S) T and*  
*atms-C: atms-of (C' + {#L#})  $\subseteq$  atms-of-mu (clauses S)  $\cup$  atm-of ' (lits-of (trail S))*  
**using** *bj-l inv backjump-l-learn-backjump n-d atm-clss atm-trail* **by** *blast*  
**have** *card-T-S: card (set-mset (clauses T))  $\leq$  1 + card (set-mset (clauses S))*  
**using** *bj-l inv* **by** (*auto elim!: backjump-lE simp: card-insert-if*)  
**have**  
 $((2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A)) (\text{trail-weight } T))$   
 $< ((2 + \text{card} (\text{atms-of-} m \ A)) \wedge (1 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-} m \ A)) (2 + \text{card} (\text{atms-of-} m \ A))$   
 $\quad (\text{trail-weight } (\text{add-cls}_{NOT} (C' + \{ \#L\# \}) S)))$   
**apply** (*rule dpll-bj-trail-mes-decreasing-prop*)  
**using** *bj bj-backjump* **apply** *blast*

using  $cdcl_{NOT}.c\text{-learn } cdcl_{NOT}.cdcl_{NOT}\text{-inv } inv \text{ learn } \mathbf{apply } \mathbf{blast}$   
 using  $atms\text{-}C \text{ atm-clss } atm\text{-trail } n\text{-d } \mathbf{apply } \mathbf{fastforce}$   
 using  $atm\text{-trail } n\text{-d } \mathbf{apply } \mathbf{simp}$   
 apply  $(simp \text{ add: } n\text{-d})$   
 using  $fin\text{-}A \mathbf{apply } \mathbf{simp}$   
 done  
 then have  $((2 + \text{card } (atms\text{-of-}m \ A)) \wedge (1 + \text{card } (atms\text{-of-}m \ A))$   
 $\quad - \mu_C (1 + \text{card } (atms\text{-of-}m \ A)) (2 + \text{card } (atms\text{-of-}m \ A)) (\text{trail-weight } T))$   
 $< ((2 + \text{card } (atms\text{-of-}m \ A)) \wedge (1 + \text{card } (atms\text{-of-}m \ A))$   
 $\quad - \mu_C (1 + \text{card } (atms\text{-of-}m \ A)) (2 + \text{card } (atms\text{-of-}m \ A)) (\text{trail-weight } S))$   
 using  $n\text{-d } \mathbf{by } \mathbf{auto}$   
 then show  $?case$   
 using  $card\text{-}T\text{-}S \text{ unfolding } \mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def } \mathbf{by } \mathbf{linarith}$   
 qed

**lemma**  $wf\text{-}cdcl_{NOT}\text{-merged-bj-learn}$ :

assumes  
 $fin\text{-}A$ :  $finite \ A$   
 shows  $wf \ \{(T, S).$   
 $(inv \ S \wedge atms\text{-of-}\mu \ (\text{clauses } S) \subseteq atms\text{-of-}m \ A \wedge atm\text{-of } ' \text{ lits-of } (\text{trail } S) \subseteq atms\text{-of-}m \ A$   
 $\wedge no\text{-dup } (\text{trail } S))$   
 $\wedge cdcl_{NOT}\text{-merged-bj-learn } S \ T\}$   
 apply  $(rule \ wfP\text{-if-measure}[of \ - \ - \ \mu_{CDCL}'\text{-merged } A])$   
 using  $cdcl_{NOT}\text{-decreasing-measure}' \ fin\text{-}A \mathbf{by } \mathbf{simp}$

**lemma**  $tranclp\text{-}cdcl_{NOT}\text{-}cdcl_{NOT}\text{-tranclp}$ :

assumes  
 $cdcl_{NOT}\text{-merged-bj-learn}^{++} \ S \ T$  and  
 $inv$ :  $inv \ S$  and  
 $atm\text{-clss}$ :  $atms\text{-of-}\mu \ (\text{clauses } S) \subseteq atms\text{-of-}m \ A$  and  
 $atm\text{-trail}$ :  $atm\text{-of } ' \text{ lits-of } (\text{trail } S) \subseteq atms\text{-of-}m \ A$  and  
 $n\text{-d}$ :  $no\text{-dup } (\text{trail } S)$  and  
 $fin\text{-}A[simp]$ :  $finite \ A$   
 shows  $(T, S) \in \{(T, S).$   
 $(inv \ S \wedge atms\text{-of-}\mu \ (\text{clauses } S) \subseteq atms\text{-of-}m \ A \wedge atm\text{-of } ' \text{ lits-of } (\text{trail } S) \subseteq atms\text{-of-}m \ A$   
 $\wedge no\text{-dup } (\text{trail } S))$   
 $\wedge cdcl_{NOT}\text{-merged-bj-learn } S \ T\}^+ \ (\mathbf{is} \ - \in ?P^+)$   
 using  $assms(1)$   
**proof**  $(induction \ rule: \ tranclp\text{-}induct)$   
 case  $base$   
 then show  $?case$  using  $n\text{-d } atm\text{-clss } atm\text{-trail } inv \mathbf{by } \mathbf{auto}$   
 next  
 case  $(step \ T \ U)$  note  $st = this(1)$  and  $cdcl_{NOT} = this(2)$  and  $IH = this(3)$   
 have  $cdcl_{NOT}^{**} \ S \ T$   
 apply  $(rule \ rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-is-rtranclp}\text{-}cdcl_{NOT})$   
 using  $st \ cdcl_{NOT} \ inv \ n\text{-d } atm\text{-clss } atm\text{-trail } inv \mathbf{by } \mathbf{auto}$   
 have  $inv \ T$   
 apply  $(rule \ rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn}\text{-}inv)$   
 using  $inv \ st \ cdcl_{NOT} \ n\text{-d } atm\text{-clss } atm\text{-trail } inv \mathbf{by } \mathbf{auto}$   
 moreover have  $atms\text{-of-}\mu \ (\text{clauses } T) \subseteq atms\text{-of-}m \ A$   
 using  $cdcl_{NOT}.rtranclp\text{-}cdcl_{NOT}\text{-trail-clauses-bound}[OF \ \langle cdcl_{NOT}^{**} \ S \ T \rangle \ inv \ n\text{-d } atm\text{-clss } atm\text{-trail}]$   
 by  $fast$   
 moreover have  $atm\text{-of } ' \ (\text{lits-of } (\text{trail } T)) \subseteq atms\text{-of-}m \ A$   
 using  $cdcl_{NOT}.rtranclp\text{-}cdcl_{NOT}\text{-trail-clauses-bound}[OF \ \langle cdcl_{NOT}^{**} \ S \ T \rangle \ inv \ n\text{-d } atm\text{-clss } atm\text{-trail}]$   
 by  $fast$

```

moreover have no-dup (trail T)
  using cdclNOT.rtrancpl-cdclNOT-no-dup[OF  $\langle \text{cdcl}_{NOT}^{**} S T \rangle \text{ inv } n\text{-d}$ ] by fast
ultimately have (U, T)  $\in ?P$ 
  using cdclNOT by auto
then show ?case using IH by (simp add: trancpl-into-trancpl2)
qed

```

```

lemma wf-trancpl-cdclNOT-merged-bj-learn:
assumes finite A
shows wf  $\{(T, S).$ 
  (inv S  $\wedge$  atms-of-mu (clauses S)  $\subseteq$  atms-of-m A  $\wedge$  atm-of 'lits-of' (trail S)  $\subseteq$  atms-of-m A
   $\wedge$  no-dup (trail S))
   $\wedge$  cdclNOT-merged-bj-learn++ S T $\}$ 
apply (rule wf-subset)
apply (rule wf-trancpl[OF wf-cdclNOT-merged-bj-learn])
using assms apply simp
using trancpl-cdclNOT-cdclNOT-trancpl[OF - - - -  $\langle \text{finite } A \rangle$ ] by auto

```

```

lemma backjump-no-step-backjump-l:
backjump S T  $\implies$  inv S  $\implies$   $\neg$ no-step backjump-l S
apply (elim backjumpE)
apply (rule bj-can-jump)
apply auto[7]
by blast

```

```

lemma cdclNOT-merged-bj-learn-final-state:
fixes A :: 'v literal multiset set' and S T :: 'st'
assumes
  n-s: no-step cdclNOT-merged-bj-learn S and
  atms-S: atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and
  atms-trail: atm-of 'lits-of' (trail S)  $\subseteq$  atms-of-m A and
  n-d: no-dup (trail S) and
  finite A and
  inv: inv S and
  decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
shows unsatisfiable (set-mset (clauses S))
   $\vee$  (trail S  $\models_{asm}$  clauses S  $\wedge$  satisfiable (set-mset (clauses S)))
proof -
let ?N = set-mset (clauses S)
let ?M = trail S
consider
  (sat) satisfiable ?N and ?M  $\models_{as}$  ?N
  | (sat') satisfiable ?N and  $\neg$  ?M  $\models_{as}$  ?N
  | (unsat) unsatisfiable ?N
by auto
then show ?thesis
proof cases
case sat' note sat = this(1) and M = this(2)
obtain C where C  $\in$  ?N and  $\neg$  ?M  $\models_a$  C using M unfolding true-annots-def by auto
obtain I :: 'v literal set' where
  I  $\models_s$  ?N and
  cons: consistent-interp I and
  tot: total-over-m I ?N and
  atm-I-N: atm-of 'I'  $\subseteq$  atms-of-m ?N
using sat unfolding satisfiable-def-min by auto

```

```

let ?I = I ∪ {P | P. P ∈ lits-of ?M ∧ atm-of P ∉ atm-of ' I}
let ?O = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-m ?N }
have cons-I': consistent-interp ?I
  using cons using ⟨no-dup ?M⟩ unfolding consistent-interp-def
  by (auto simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def
    dest!: no-dup-cannot-not-lit-and-uminus)
have tot-I': total-over-m ?I (?N ∪ (λa. {#lit-of a#})) ' set ?M
  using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
  by fastforce
have {P | P. P ∈ lits-of ?M ∧ atm-of P ∉ atm-of ' I} ⊨s ?O
  using ⟨I ⊨s ?N⟩ atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)
then have I'-N: ?I ⊨s ?N ∪ ?O
  using ⟨I ⊨s ?N⟩ true-clss-union-increase by force
have tot': total-over-m ?I (?N ∪ ?O)
  using atm-I-N tot unfolding total-over-m-def total-over-set-def
  by (force simp: image-iff lits-of-def dest!: is-marked-ex-Marked)

have atms-N-M: atms-of-m ?N ⊆ atm-of ' lits-of ?M
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain l :: 'v where
    l-N: l ∈ atms-of-m ?N and
    l-M: l ∉ atm-of ' lits-of ?M
  by auto
  have undefined-lit ?M (Pos l)
    using l-M by (metis Marked-Propagated-in-iff-in-lits-of
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
  have decideNOT S (prepend-trail (Marked (Pos l) ()) S)
    by (metis ⟨undefined-lit ?M (Pos l)⟩ decideNOT.intros l-N literal.sel(1)
      state-eqNOT-ref)
  then show False
    using cdclNOT-merged-bj-learn-decideNOT n-s by blast
qed

have ?M ⊨as CNot C
  by (metis atms-N-M ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ all-variables-defined-not-imply-cnot
    atms-of-atms-of-m-mono atms-of-m-CNot-atms-of atms-of-m-CNot-atms-of-m subsetCE)
have ∃ l ∈ set ?M. is-marked l
proof (rule ccontr)
  let ?O = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-m ?N }
  have ∅[iff]: ∧ I. total-over-m I (?N ∪ ?O ∪ (λa. {#lit-of a#})) ' set ?M
    ↔ total-over-m I (?N ∪ (λa. {#lit-of a#})) ' set ?M
  unfolding total-over-set-def total-over-m-def atms-of-m-def by auto
  assume ¬ ?thesis
  then have [simp]: { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M }
    = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-m ?N }
  by auto
  then have ?N ∪ ?O ⊨ps (λa. {#lit-of a#}) ' set ?M
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

  then have ?I ⊨s (λa. {#lit-of a#}) ' set ?M
    using cons-I' I'-N tot-I' ⟨?I ⊨s ?N ∪ ?O⟩ unfolding ∅ true-clss-clss-def by blast
  then have lits-of ?M ⊆ ?I
    unfolding true-clss-def lits-of-def by auto
  then have ?M ⊨as ?N

```

```

    using  $I'-N \langle C \in ?N \rangle \langle \neg ?M \models_a C \rangle \text{ cons-}I' \text{ atms-}N-M$ 
    by (meson (trail  $S \models_{as} CNot\ C$ ) consistent- $CNot$ -not rev-subsetD sup-ge1 true-annot-def
        true-annots-def true-clss-mono-set-mset-l true-clss-def)
    then show False using M by fast
qed
from List.split-list-first-propE[OF this] obtain  $K :: 'v \text{ literal}$  and  $d :: unit$  and
 $F\ F' :: ('v, unit, unit) \text{ marked-lit list}$  where
 $M-K: ?M = F' @ \text{Marked } K\ () \# F$  and
 $nm: \forall f \in \text{set } F'. \neg \text{is-marked } f$ 
    unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let  $?K = \text{Marked } K\ () :: ('v, unit, unit) \text{ marked-lit}$ 
have  $?K \in \text{set } ?M$ 
    unfolding M-K by auto
let  $?C = \text{image-mset lit-of } \{\#L \in \#mset\ ?M. \text{is-marked } L \wedge L \neq ?K\# \} :: 'v \text{ literal multiset}$ 
let  $?C' = \text{set-mset } (\text{image-mset } (\lambda L :: 'v \text{ literal}. \{\#L\# \})\ (?C + \{\#lit-of\ ?K\# \}))$ 
have  $?N \cup \{\{\#lit-of\ L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M\} \models_{ps} (\lambda a. \{\#lit-of\ a\# \}) \text{ ' set } ?M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have  $C': ?C' = \{\{\#lit-of\ L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M\}$ 
    unfolding M-K apply standard
    apply force
    using IntI by auto
ultimately have  $N-C-M: ?N \cup ?C' \models_{ps} (\lambda a. \{\#lit-of\ a\# \}) \text{ ' set } ?M$ 
    by auto
have  $N-M-False: ?N \cup (\lambda L. \{\#lit-of\ L\# \}) \text{ ' (set } ?M) \models_{ps} \{\{\#\}\}$ 
    using M  $\langle ?M \models_{as} CNot\ C \rangle \langle C \in ?N \rangle$  unfolding true-clss-clss-def true-annots-def Ball-def
    true-annot-def by (metis consistent- $CNot$ -not sup.orderE sup-commute true-clss-def
        true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using  $\langle no\text{-}dup\ ?M \rangle$  unfolding M-K by (simp add: defined-lit-map)
moreover
    have  $?N \cup ?C' \models_{ps} \{\{\#\}\}$ 
    proof -
        have  $A: ?N \cup ?C' \cup (\lambda a. \{\#lit-of\ a\# \}) \text{ ' set } ?M =$ 
             $?N \cup (\lambda a. \{\#lit-of\ a\# \}) \text{ ' set } ?M$ 
            unfolding M-K by auto
        show ?thesis
            using true-clss-clss-left-right[OF N-C-M, of  $\{\{\#\}\}$ ] N-M-False unfolding A by auto
    qed
have  $?N \models_p \text{image-mset uminus } ?C + \{\#-K\# \}$ 
    unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
    proof (intro allI impI)
        fix I
        assume
            tot: total-over-set I (atms-of-m ( $?N \cup \{\text{image-mset uminus } ?C + \{\#-K\# \}\}$ )) and
            cons: consistent-interp I and
            I  $\models_s ?N$ 
        have  $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$ 
            using cons tot unfolding consistent-interp-def by (cases K) auto
        have tot': total-over-set I
            (atm-of ' lit-of ' (set  $?M \cap \{L. \text{is-marked } L \wedge L \neq \text{Marked } K\ ()\}$ ))
            using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
        { fix  $x :: ('v, unit, unit) \text{ marked-lit}$ 
            assume
                a3: lit-of  $x \notin I$  and
                a1:  $x \in \text{set } ?M$  and

```

```

    a4: is-marked x and
    a5: x ≠ Marked K ()
  then have Pos (atm-of (lit-of x)) ∈ I ∨ Neg (atm-of (lit-of x)) ∈ I
    using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
  moreover have f6: Neg (atm-of (lit-of x)) = − Pos (atm-of (lit-of x))
    by simp
  ultimately have − lit-of x ∈ I
    using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1))
} note H = this

have ¬I ⊨s ?C'
  using ⟨?N ∪ ?C' ⊨ps {{#}}⟩ tot cons ⟨I ⊨s ?N⟩
  unfolding true-clss-clss-def total-over-m-def
  by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-m-single-image-atm-of-lit-of)
then show I ⊨ image-mset uminus ?C + {# − K#}
  unfolding true-clss-def true-cl-def Bex-mset-def
  using ⟨(K ∈ I ∧ −K ∉ I) ∨ (−K ∈ I ∧ K ∉ I)⟩
  by (auto dest!: H)
qed
moreover have F ⊨as CNot (image-mset uminus ?C)
  using nm unfolding true-annots-def CNot-def M-K by (auto simp add: lits-of-def)
ultimately have False
  using bj-can-jump[of S F' K F C −K
    image-mset uminus (image-mset lit-of {# L :# mset ?M. is-marked L ∧ L ≠ Marked K ()#})]
    ⟨C ∈ ?N⟩ n-s ⟨?M ⊨as CNot C⟩ bj-backjump inv unfolding M-K
  by (auto simp: cdclNOT-merged-bj-learn.simps)
then show ?thesis by fast
qed auto
qed

lemma full-cdclNOT-merged-bj-learn-final-state:
  fixes A :: 'v literal multiset set and S T :: 'st
  assumes
    full: full cdclNOT-merged-bj-learn S T and
    atms-S: atms-of-mu (clauses S) ⊆ atms-of-m A and
    atms-trail: atm-of ' lits-of (trail S) ⊆ atms-of-m A and
    n-d: no-dup (trail S) and
    finite A and
    inv: inv S and
    decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
  shows unsatisfiable (set-mset (clauses T))
    ∨ (trail T ⊨asm clauses T ∧ satisfiable (set-mset (clauses T)))
proof −
  have st: cdclNOT-merged-bj-learn** S T and n-s: no-step cdclNOT-merged-bj-learn T
    using full unfolding full-def by blast+
  then have st: cdclNOT** S T
    using inv rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT-and-inv n-d by auto
  have atms-of-mu (clauses T) ⊆ atms-of-m A and atm-of ' lits-of (trail T) ⊆ atms-of-m A
    using cdclNOT.rtranclp-cdclNOT-trail-clauses-bound[OF st inv n-d atms-S atms-trail] by blast+
  moreover have no-dup (trail T)
    using cdclNOT.rtranclp-cdclNOT-no-dup inv n-d st by blast
  moreover have inv T
    using cdclNOT.rtranclp-cdclNOT-inv inv st by blast
  moreover have all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))

```

```

    using cdclNOT.rtrancp-cdclNOT-all-decomposition-implies inv st decomp n-d by blast
ultimately show ?thesis
    using cdclNOT-merged-bj-learn-final-state[of T A] (finite A) n-s by fast
qed

end

```

### 14.8.1 Instantiations

```

locale cdclNOT-with-backtrack-and-restarts =
  conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt trail clauses
  prepend-trail tl-trail add-clsNOT remove-clsNOT propagate-cons inv backjump-cons
  learn-restrictions forget-restrictions
for
  trail :: 'st  $\Rightarrow$  ('v::linorder, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v::linorder clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT remove-clsNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-cons :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-cons :: 'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  learn-restrictions forget-restrictions :: 'v::linorder clause  $\Rightarrow$  'st  $\Rightarrow$  bool
  +
fixes f :: nat  $\Rightarrow$  nat
assumes
  unbounded: unbounded f and f-ge-1:  $\bigwedge n. n \geq 1 \Rightarrow f\ n \geq 1$  and
  inv-restart:  $\bigwedge S\ T. inv\ S \Rightarrow T \sim reduce-trail-to_{NOT} \sqcup S \Rightarrow inv\ T$ 
begin

lemma bound-inv-inv:
assumes
  inv S and
  n-d: no-dup (trail S) and
  atms-clss-S-A: atms-of-mu (clauses S)  $\subseteq$  atms-of-m A and
  atms-trail-S-A: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-m A and
  finite A and
  cdclNOT: cdclNOT S T
shows
  atms-of-mu (clauses T)  $\subseteq$  atms-of-m A and
  atm-of ' lits-of (trail T)  $\subseteq$  atms-of-m A and
  finite A
proof –
  have cdclNOT S T
  using (inv S) cdclNOT by linarith
  then have atms-of-mu (clauses T)  $\subseteq$  atms-of-mu (clauses S)  $\cup$  atm-of ' lits-of (trail S)
  using (inv S)
  by (meson conflict-driven-clause-learning-ops.cdclNOT-atms-of-m-clauses-decreasing
    conflict-driven-clause-learning-ops-axioms n-d)
  then show atms-of-mu (clauses T)  $\subseteq$  atms-of-m A
  using atms-clss-S-A atms-trail-S-A by blast
next
  show atm-of ' lits-of (trail T)  $\subseteq$  atms-of-m A
  by (meson (inv S) atms-clss-S-A atms-trail-S-A cdclNOT cdclNOT-atms-in-trail-in-set n-d)
next
  show finite A

```

using  $\langle \text{finite } A \rangle$  by *simp*  
 qed  
 sublocale *cdcl<sub>NOT</sub>-increasing-restarts-ops*  $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} \parallel S \text{ cdcl}_{NOT} f$   
 $\lambda A S. \text{atms-of-mu } (\text{clauses } S) \subseteq \text{atms-of-m } A \wedge \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-m } A \wedge$   
 $\text{finite } A$   
 $\mu_{CDCL}' \lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$   
 $\mu_{CDCL}'\text{-bound}$   
 apply *unfold-locales*  
 apply (*simp add: unbounded*)  
 using *f-ge-1* apply *force*  
 using *bound-inv-inv* apply *meson*  
 apply (*rule cdcl<sub>NOT</sub>-decreasing-measure'; simp*)  
 apply (*rule rtranclp-cdcl<sub>NOT</sub>- $\mu_{CDCL}'$ -bound; simp*)  
 apply (*rule rtranclp- $\mu_{CDCL}'$ -bound-decreasing; simp*)  
 apply *auto*[]  
 apply *auto*[]  
 using *cdcl<sub>NOT</sub>-inv cdcl<sub>NOT</sub>-no-dup* apply *blast*  
 using *inv-restart* apply *auto*[]  
 done

**abbreviation** *cdcl<sub>NOT</sub>-l* where

*cdcl<sub>NOT</sub>-l*  $\equiv$   
*conflict-driven-clause-learning-ops.cdcl<sub>NOT</sub> trail clauses prepend-trail tl-trail add-cl<sub>NOT</sub>*  
*remove-cl<sub>NOT</sub> propagate-conds* ( $\lambda - S T. \text{backjump } S T$ )  
 $(\lambda C S. \text{distinct-mset } C \wedge \neg \text{tautology } C \wedge \text{learn-restrictions } C S$   
 $\wedge (\exists F K F' C' L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge C = C' + \{\#L\# \}$   
 $\wedge F \models_{as} C \text{Not } C' \wedge C' + \{\#L\# \} \notin \text{clauses } S))$   
 $(\lambda C S. \neg (\exists F' F K L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } (C - \{\#L\# \})))$   
 $\wedge \text{forget-restrictions } C S)$

**lemma** *cdcl<sub>NOT</sub>-with-restart- $\mu_{CDCL}'$ -le- $\mu_{CDCL}'$ -bound:*

assumes  
 $\text{cdcl}_{NOT}: \text{cdcl}_{NOT}\text{-restart } (T, a) (V, b)$  and  
 $\text{cdcl}_{NOT}\text{-inv:}$   
 $\text{inv } T$   
 $\text{no-dup } (\text{trail } T)$  and  
 $\text{bound-inv:}$   
 $\text{atms-of-mu } (\text{clauses } T) \subseteq \text{atms-of-m } A$   
 $\text{atm-of } ' \text{ lits-of } (\text{trail } T) \subseteq \text{atms-of-m } A$   
 $\text{finite } A$   
 shows  $\mu_{CDCL}' A V \leq \mu_{CDCL}'\text{-bound } A T$   
 using *cdcl<sub>NOT</sub>-inv bound-inv*  
**proof** (*induction rule: cdcl<sub>NOT</sub>-with-restart-induct[OF cdcl<sub>NOT</sub>]*)  
**case** ( $1 m S T n U$ ) **note**  $U = \text{this}(3)$   
**show** ?*case*  
 apply (*rule rtranclp-cdcl<sub>NOT</sub>- $\mu_{CDCL}'$ -bound-reduce-trail-to<sub>NOT</sub>[of  $S T$ ]*)  
 using  $\langle (\text{cdcl}_{NOT} \rightsquigarrow m) S T \rangle$  apply (*fastforce dest!: relpowp-imp-rtranclp*)  
 using  $1$  by *auto*  
**next**  
**case** ( $2 S T n$ ) **note**  $\text{full} = \text{this}(2)$   
**show** ?*case*  
 apply (*rule rtranclp-cdcl<sub>NOT</sub>- $\mu_{CDCL}'$ -bound*)  
 using  $\text{full } 2$  **unfolding** *full1-def* by *force+*  
 qed



**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-mu* (*clauses* *T*)  $\subseteq$  *atms-of-m* *A*  
*atm-of* ‘*lits-of*’ (*trail* *T*)  $\subseteq$  *atms-of-m* *A*  
*finite* *A*  
**shows**  $\mu_{CDCL}'\text{-bound } A \ V \leq \mu_{CDCL}'\text{-bound } A \ T$   
**using** *cdcl<sub>NOT</sub>-inv* *bound-inv*  
**proof** (*induction rule:* *cdcl<sub>NOT</sub>-with-restart-induct*[*OF cdcl<sub>NOT</sub>*])  
**case** (*1 m S T n U*) **note** *U = this(3)*  
**have**  $\mu_{CDCL}'\text{-bound } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$   
**apply** (*rule* *rtrancp- $\mu_{CDCL}'$ -bound-decreasing*)  
**using**  $\langle (cdcl_{NOT} \rightsquigarrow m) \ S \ T \rangle$  **apply** (*fastforce* *dest: relpowp-imp-rtrancp*)  
**using** *1* **by** *auto*  
**then show** *?case* **using** *U* **unfolding**  $\mu_{CDCL}'\text{-bound-def}$  **by** *auto*  
**next**  
**case** (*2 S T n*) **note** *full = this(2)*  
**show** *?case*  
**apply** (*rule* *rtrancp- $\mu_{CDCL}'$ -bound-decreasing*)  
**using** *full 2* **unfolding** *full1-def* **by** *force+*  
**qed**

**sublocale** *cdcl<sub>NOT</sub>-increasing-restarts - - - - - f*  
 $\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} \ \square \ S$   
 $\lambda A \ S. \ \text{atms-of-mu} \ (\text{clauses } S) \subseteq \text{atms-of-m } A$   
 $\wedge \text{atm-of 'lits-of' } (\text{trail } S) \subseteq \text{atms-of-m } A \wedge \text{finite } A$   
 $\mu_{CDCL}' \text{ cdcl}_{NOT}$   
 $\lambda S. \ \text{inv } S \wedge \text{no-dup } (\text{trail } S)$   
 $\mu_{CDCL}'\text{-bound}$   
**apply** *unfold-locales*  
**using** *cdcl<sub>NOT</sub>-with-restart- $\mu_{CDCL}'$ -le- $\mu_{CDCL}'$ -bound* **apply** *simp*  
**using** *cdcl<sub>NOT</sub>-with-restart- $\mu_{CDCL}'$ -bound-le- $\mu_{CDCL}'$ -bound* **apply** *simp*  
**done**

**lemma** *cdcl<sub>NOT</sub>-restart-all-decomposition-implies:*  
**assumes** *cdcl<sub>NOT</sub>-restart* *S T* **and**  
*inv* (*fst* *S*) **and**  
*no-dup* (*trail* (*fst* *S*))  
*all-decomposition-implies-m* (*clauses* (*fst* *S*)) (*get-all-marked-decomposition* (*trail* (*fst* *S*)))  
**shows**  
*all-decomposition-implies-m* (*clauses* (*fst* *T*)) (*get-all-marked-decomposition* (*trail* (*fst* *T*)))  
**using** *assms* **apply** (*induction*)  
**using** *rtrancp-cdcl<sub>NOT</sub>-all-decomposition-implies* **by** (*auto* *dest!:* *trancp-into-rtrancp*  
*simp: full1-def*)

**lemma** *rtrancp-cdcl<sub>NOT</sub>-restart-all-decomposition-implies:*  
**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* (*fst* *S*)) (*get-all-marked-decomposition* (*trail* (*fst* *S*)))

**shows**  
*all-decomposition-implies-m* (*clauses* (*fst T*)) (*get-all-marked-decomposition* (*trail* (*fst T*)))  
**using** *assms*(1)  
**proof** (*induction rule: rtrancpl-induct*)  
**case** *base*  
**then show** ?*case* **using** *decomp* **by** *simp*  
**next**  
**case** (*step T u*) **note** *st = this(1)* **and** *r = this(2)* **and** *IH = this(3)*  
**have** *inv* (*fst T*)  
**using** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv*[*OF st*] *inv n-d* **by** *blast*  
**moreover have** *no-dup* (*trail* (*fst T*))  
**using** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv*[*OF st*] *inv n-d* **by** *blast*  
**ultimately show** ?*case*  
**using** *cdcl<sub>NOT</sub>-restart-all-decomposition-implies r IH n-d* **by** *fast*  
**qed**

**lemma** *cdcl<sub>NOT</sub>-restart-sat-ext-iff*:  
**assumes**  
*st: cdcl<sub>NOT</sub>-restart S T* **and**  
*n-d: no-dup* (*trail* (*fst S*)) **and**  
*inv: inv* (*fst S*)  
**shows**  $I \models_{\text{sextm}} \text{clauses}(\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}(\text{fst } T)$   
**using** *assms*  
**proof** (*induction*)  
**case** (*restart-step m S T n U*)  
**then show** ?*case*  
**using** *rtrancpl-cdcl<sub>NOT</sub>-bj-sat-ext-iff n-d* **by** (*fastforce dest!: relpowp-imp-rtrancpl*)  
**next**  
**case** *restart-full*  
**then show** ?*case* **using** *rtrancpl-cdcl<sub>NOT</sub>-bj-sat-ext-iff* **unfolding** *full1-def*  
**by** (*fastforce dest!: trancpl-into-rtrancpl*)  
**qed**

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-restart-sat-ext-iff*:  
**assumes**  
*st: cdcl<sub>NOT</sub>-restart\*\* S T* **and**  
*n-d: no-dup* (*trail* (*fst S*)) **and**  
*inv: inv* (*fst S*)  
**shows**  $I \models_{\text{sextm}} \text{clauses}(\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}(\text{fst } T)$   
**using** *st*  
**proof** (*induction*)  
**case** *base*  
**then show** ?*case* **by** *simp*  
**next**  
**case** (*step T U*) **note** *st = this(1)* **and** *r = this(2)* **and** *IH = this(3)*  
**have** *inv* (*fst T*)  
**using** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv*[*OF st*] *inv n-d* **by** *blast*+  
**moreover have** *no-dup* (*trail* (*fst T*))  
**using** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv* *rtrancpl-cdcl<sub>NOT</sub>-no-dup st inv n-d* **by** *blast*  
**ultimately show** ?*case*  
**using** *cdcl<sub>NOT</sub>-restart-sat-ext-iff*[*OF r*] *IH* **by** *blast*  
**qed**

**theorem** *full-cdcl<sub>NOT</sub>-restart-backjump-final-state*:  
**fixes** *A :: 'v literal multiset set* **and** *S T :: 'st*

**assumes**  
*full*: *full cdcl<sub>NOT</sub>-restart* (*S*, *n*) (*T*, *m*) **and**  
*atms-S*: *atms-of-mu* (*clauses S*)  $\subseteq$  *atms-of-m A* **and**  
*atms-trail*: *atm-of* ‘*lits-of* (*trail S*)  $\subseteq$  *atms-of-m 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 S*) (*get-all-marked-decomposition* (*trail S*))  
**shows** *unsatisfiable* (*set-mset* (*clauses S*))  
 $\vee$  (*lits-of* (*trail T*)  $\models_{\text{sextm}}$  *clauses S*  $\wedge$  *satisfiable* (*set-mset* (*clauses S*)))  
**proof** –  
**have** *st*: *cdcl<sub>NOT</sub>-restart\*\** (*S*, *n*) (*T*, *m*) **and**  
*n-s*: *no-step cdcl<sub>NOT</sub>-restart* (*T*, *m*)  
**using** *full unfolding full-def* **by** *fast+*  
**have** *binv-T*: *atms-of-mu* (*clauses T*)  $\subseteq$  *atms-of-m A* *atm-of* ‘*lits-of* (*trail T*)  $\subseteq$  *atms-of-m A*  
**using** *rtranclp-cdcl<sub>NOT</sub>-with-restart-bound-inv*[*OF st*, *of A*] *inv n-d atms-S atms-trail*  
**by** *auto*  
**moreover have** *inv-T*: *no-dup* (*trail T*) *inv T*  
**using** *rtranclp-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv*[*OF st*] *inv n-d* **by** *auto*  
**moreover have** *all-decomposition-implies-m* (*clauses T*) (*get-all-marked-decomposition* (*trail T*))  
**using** *rtranclp-cdcl<sub>NOT</sub>-restart-all-decomposition-implies*[*OF st*] *inv n-d*  
*decomp* **by** *auto*  
**ultimately have** *T*: *unsatisfiable* (*set-mset* (*clauses T*))  
 $\vee$  (*trail T*  $\models_{\text{asm}}$  *clauses T*  $\wedge$  *satisfiable* (*set-mset* (*clauses T*)))  
**using** *no-step-cdcl<sub>NOT</sub>-restart-no-step-cdcl<sub>NOT</sub>*[*of* (*T*, *m*) *A*] *n-s*  
*cdcl<sub>NOT</sub>-final-state*[*of T A*] **unfolding** *cdcl<sub>NOT</sub>-NOT-all-inv-def* **by** *auto*  
**have** *eq-sat-S-T*:  $\bigwedge I. I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$   
**using** *rtranclp-cdcl<sub>NOT</sub>-restart-sat-ext-iff*[*OF st*] *inv n-d atms-S*  
*atms-trail* **by** *auto*  
**have** *cons-T*: *consistent-interp* (*lits-of* (*trail T*))  
**using** *inv-T(1) distinctconsistent-interp* **by** *blast*  
**consider**  
(*unsat*) *unsatisfiable* (*set-mset* (*clauses T*))  
| (*sat*) *trail T*  $\models_{\text{asm}}$  *clauses T* **and** *satisfiable* (*set-mset* (*clauses T*))  
**using** *T* **by** *blast*  
**then show** *?thesis*  
**proof** *cases*  
**case** *unsat*  
**then have** *unsatisfiable* (*set-mset* (*clauses S*))  
**using** *eq-sat-S-T consistent-true-clss-ext-satisfiable true-clss-imp-true-cls-ext*  
**unfolding** *satisfiable-def* **by** *blast*  
**then show** *?thesis* **by** *fast*  
**next**  
**case** *sat*  
**then have** *lits-of* (*trail T*)  $\models_{\text{sextm}}$  *clauses S*  
**using** *rtranclp-cdcl<sub>NOT</sub>-restart-sat-ext-iff*[*OF st*] *inv n-d atms-S*  
*atms-trail* **by** (*auto simp: true-clss-imp-true-cls-ext true-annots-true-cls*)  
**moreover then have** *satisfiable* (*set-mset* (*clauses S*))  
**using** *cons-T consistent-true-clss-ext-satisfiable* **by** *blast*  
**ultimately show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**end** — end of *cdcl<sub>NOT</sub>-with-backtrack-and-restarts* locale

locale *most-general-cdcl<sub>NOT</sub>* =

```

dpll-state trail clauses prepend-trail tl-trail add-clNOT remove-clNOT +
propagate-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds +
backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT λ- - - -. True
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clNOT remove-clNOT:: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool
begin
lemma backjump-bj-can-jump:
  assumes
    tr-S: trail S = F' @ Marked K () # F and
    C: C ∈# clauses S and
    tr-S-C: trail S ⊨as CNot C and
    undef: undefined-lit F L and
    atm-L: atm-of L ∈ atms-of-mu (clauses S) ∪ atm-of (' (lits-of (F' @ Marked K () # F)) and
    cls-S-C': clauses S ⊨pm C' + {#L#} and
    F-C': F ⊨as CNot C'
  shows ¬no-step backjump S
  using backjump.intros[OF tr-S - C tr-S-C undef - cls-S-C' F-C',
    of prepend-trail (Propagated L -) (reduce-trail-toNOT F S)] atm-L unfolding tr-S
  by (auto simp: state-eqNOT-def simp del: state-simpNOT)

sublocale dpll-with-backjumping-ops - - - - - inv λ- - - -. True
  using backjump-bj-can-jump by unfold-locales auto
end

The restart does only reset the trail, contrary to Weidenbach's version. But there is a forget
rule.

locale cdclNOT-merge-bj-learn-with-backtrack-restarts =
  cdclNOT-merge-bj-learn trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv forget-conds
  λC L S. distinct-mset (C + {#L#}) ∧ backjump-l-cond C L S
  for
    trail :: 'st ⇒ ('v::linorder, unit, unit) marked-lits and
    clauses :: 'st ⇒ 'v::linorder clauses and
    prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
    tl-trail :: 'st ⇒ 'st and
    add-clNOT remove-clNOT:: 'v clause ⇒ 'st ⇒ 'st and
    propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
    inv :: 'st ⇒ bool and
    forget-conds :: 'v clause ⇒ 'st ⇒ bool and
    backjump-l-cond :: 'v clause ⇒ 'v literal ⇒ 'st ⇒ bool
  +
  fixes f :: nat ⇒ nat
  assumes
    unbounded: unbounded f and f-ge-1: ∧n. n ≥ 1 ⇒ f n ≥ 1 and
    inv-restart: ∧S T. inv S ⇒ T ∼ reduce-trail-toNOT [] S ⇒ inv T
begin

```

**interpretation**  $cdcl_{NOT}$ :

*conflict-driven-clause-learning-ops* *trail clauses prepend-trail tl-trail add-cls<sub>NOT</sub> remove-cls<sub>NOT</sub>*  
*propagate-conds inv backjump-conds* ( $\lambda C \cdot \text{distinct-mset } C \wedge \neg \text{tautology } C$ ) *forget-conds*  
**by** *unfold-locales*

**interpretation**  $cdcl_{NOT}$ :

*conflict-driven-clause-learning* *trail clauses prepend-trail tl-trail add-cls<sub>NOT</sub> remove-cls<sub>NOT</sub>*  
*propagate-conds inv backjump-conds* ( $\lambda C \cdot \text{distinct-mset } C \wedge \neg \text{tautology } C$ ) *forget-conds*  
**apply** *unfold-locales*  
**using**  $cdcl_{NOT}\text{-merged-bj-learn-forget}_{NOT}$   $cdcl\text{-merged-inv}$  *learn-inv*  
**by** (*auto simp add:  $cdcl_{NOT}.\text{sims}$  dpll-bj-inv*)

**definition** *not-simplified-cls*  $A = \{\#C \in \# A. \text{tautology } C \vee \neg \text{distinct-mset } C\}$

**lemma** *build-all-simple-clss-or-not-simplified-cls*:

**assumes** *atms-of-mu* (*clauses*  $S$ )  $\subseteq$  *atms-of-m*  $A$  **and**

$x \in \# \text{clauses } S$  **and** *finite*  $A$

**shows**  $x \in \text{build-all-simple-clss} (\text{atms-of-m } A) \vee x \in \# \text{not-simplified-cls} (\text{clauses } S)$

**proof** –

**consider**

(*simpl*)  $\neg \text{tautology } x$  **and** *distinct-mset*  $x$

| (*n-simp*) *tautology*  $x \vee \neg \text{distinct-mset } x$

**by** *auto*

**then show** *?thesis*

**proof** *cases*

**case** *simpl*

**then have**  $x \in \text{build-all-simple-clss} (\text{atms-of-m } A)$

**by** (*meson* *assms* *atms-of-atms-of-m-mono* *atms-of-m-finite* *build-all-simple-clss-mono*  
*distinct-mset-not-tautology-implies-in-build-all-simple-clss* *finite-subset*  
*mem-set-mset-iff subsetCE*)

**then show** *?thesis* **by** *blast*

**next**

**case** *n-simp*

**then have**  $x \in \# \text{not-simplified-cls} (\text{clauses } S)$

**using** ( $x \in \# \text{clauses } S$ ) **unfolding** *not-simplified-cls-def* **by** *auto*

**then show** *?thesis* **by** *blast*

**qed**

**qed**

**lemma**  $cdcl_{NOT}\text{-merged-bj-learn-clauses-bound}$ :

**assumes**

$cdcl_{NOT}\text{-merged-bj-learn } S T$  **and**

*inv: inv*  $S$  **and**

*atms-clss: atms-of-mu* (*clauses*  $S$ )  $\subseteq$  *atms-of-m*  $A$  **and**

*atms-trail: atm-of* (*lits-of* (*trail*  $S$ ))  $\subseteq$  *atms-of-m*  $A$  **and**

*n-d: no-dup* (*trail*  $S$ ) **and**

*fin-A[simp]: finite*  $A$

**shows** *set-mset* (*clauses*  $T$ )  $\subseteq$  *set-mset* (*not-simplified-cls* (*clauses*  $S$ ))  
 $\cup \text{build-all-simple-clss} (\text{atms-of-m } A)$

**using** *assms*

**proof** (*induction rule:  $cdcl_{NOT}\text{-merged-bj-learn.induct}$* )

**case**  $cdcl_{NOT}\text{-merged-bj-learn-decide}_{NOT}$

**then show** *?case* **using** *dpll-bj-clauses* **by** (*force dest!: build-all-simple-clss-or-not-simplified-cls*)

**next**

**case**  $cdcl_{NOT}\text{-merged-bj-learn-propagate}_{NOT}$

**then show** *?case using dpll-bj-clauses by (force dest!: build-all-simple-clss-or-not-simplified-cls)*  
**next**  
**case** *cdcl<sub>NOT</sub>-merged-bj-learn-forget<sub>NOT</sub>*  
**then show** *?case using clauses-remove-cl<sub>NOT</sub> unfolding state-eq<sub>NOT</sub>-def*  
*by (force elim!: forgetE dest: build-all-simple-clss-or-not-simplified-cls)*  
**next**  
**case** *(cdcl<sub>NOT</sub>-merged-bj-learn-backjump-l T) note bj = this(1) and inv = this(2) and*  
*atms-clss = this(3) and atms-trail = this(4) and n-d = this(5)*  
  
**have** *cdcl<sub>NOT</sub>\*\* S T*  
**apply** *(rule rtrancpl-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtrancpl-cdcl<sub>NOT</sub>)*  
**using** *(backjump-l S T) inv cdcl<sub>NOT</sub>-merged-bj-learn.simps n-d by blast+*  
**have** *atm-of (lits-of (trail T)) ⊆ atms-of-m A*  
**using** *cdcl<sub>NOT</sub>.rtrancpl-cdcl<sub>NOT</sub>-trail-clauses-bound[OF (cdcl<sub>NOT</sub>\*\* S T) inv atms-trail atms-clss*  
*n-d by auto*  
**have** *atms-of-mu (clauses T) ⊆ atms-of-m A*  
**using** *cdcl<sub>NOT</sub>.rtrancpl-cdcl<sub>NOT</sub>-trail-clauses-bound[OF (cdcl<sub>NOT</sub>\*\* S T) inv n-d atms-clss atms-trail]*  
**by fast**  
**moreover have** *no-dup (trail T)*  
**using** *cdcl<sub>NOT</sub>.rtrancpl-cdcl<sub>NOT</sub>-no-dup[OF (cdcl<sub>NOT</sub>\*\* S T) inv n-d] by fast*  
  
**obtain** *F' K F L l C' C where*  
*tr-S: trail S = F' @ Marked K () # F and*  
*T: T ~ prepend-trail (Propagated L l) (reduce-trail-to<sub>NOT</sub> F (add-cl<sub>NOT</sub> (C' + {#L#}) S)) and*  
*C ∈# clauses S and*  
*trail S ⊨<sub>as</sub> CNot C and*  
*undef: undefined-lit F L and*  
*atm-of L = atm-of K ∨ atm-of L ∈ atms-of-mu (clauses S)*  
*∨ atm-of L ∈ atm-of (lits-of F' ∪ lits-of F) and*  
*clauses S ⊨<sub>pm</sub> C' + {#L#} and*  
*F ⊨<sub>as</sub> CNot C' and*  
*dist: distinct-mset (C' + {#L#}) and*  
*tauto: ¬ tautology (C' + {#L#}) and*  
*backjump-l-cond C' L T*  
**using** *(backjump-l S T) apply (induction rule: backjump-l.induct) by auto*  
  
**have** *atms-of C' ⊆ atm-of (lits-of F)*  
**using** *(F ⊨<sub>as</sub> CNot C') by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*  
*atms-of-def image-subset-iff in-CNot-implies-uminus(2))*  
**then have** *atms-of (C' + {#L#}) ⊆ atms-of-m A*  
**using** *T (atm-of (lits-of (trail T) ⊆ atms-of-m A) tr-S undef n-d by auto*  
**then have** *build-all-simple-clss (atms-of (C' + {#L#})) ⊆ build-all-simple-clss (atms-of-m A)*  
**apply – by** *(rule build-all-simple-clss-mono) (simp-all)*  
**then have** *C' + {#L#} ∈ build-all-simple-clss (atms-of-m A)*  
**using** *distinct-mset-not-tautology-implies-in-build-all-simple-clss[OF dist tauto]*  
**by auto**  
**then show** *?case*  
**using** *T inv atms-clss undef tr-S n-d by (auto dest!: build-all-simple-clss-or-not-simplified-cls)*

qed

**lemma** *cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing:*  
**assumes** *cdcl<sub>NOT</sub>-merged-bj-learn S T*  
**shows** *(not-simplified-cl (clauses T)) ⊆# (not-simplified-cl (clauses S))*  
**using** *assms apply induction*

**prefer** 4  
**unfolding** *not-simplified-cls-def* **apply** (auto elim!: backjump-LE forgetE)[3]  
**by** (elim backjump-LE) auto

**lemma** *rtrancp-cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing*:  
**assumes** *cdcl<sub>NOT</sub>-merged-bj-learn\*\* S T*  
**shows** (*not-simplified-cls (clauses T)*)  $\subseteq \#$  (*not-simplified-cls (clauses S)*)  
**using** *assms* **apply** *induction*  
**apply** *simp*  
**by** (*drule cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing*) auto

**lemma** *rtrancp-cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound*:

**assumes**  
*cdcl<sub>NOT</sub>-merged-bj-learn\*\* S T* **and**  
*inv S* **and**  
*atms-of-mu (clauses S)  $\subseteq$  atms-of-m A* **and**  
*atm-of ' (lits-of (trail S))  $\subseteq$  atms-of-m A* **and**  
*n-d: no-dup (trail S)* **and**  
*finite[simp]: finite A*  
**shows** *set-mset (clauses T)  $\subseteq$  set-mset (not-simplified-cls (clauses S))*  
 $\cup$  *build-all-simple-clss (atms-of-m A)*  
**using** *assms(1-5)*  
**proof** *induction*  
**case** *base*  
**then show** ?*case* **by** (auto dest!: *build-all-simple-clss-or-not-simplified-cls*)

**next**

**case** (*step T U*) **note** *st = this(1)* **and** *cdcl<sub>NOT</sub> = this(2)* **and** *IH = this(3)[OF this(4-7)]* **and**  
*inv = this(4)* **and** *atms-clss-S = this(5)* **and** *atms-trail-S = this(6)* **and** *finite-clss-S = this(7)*  
**have** *st': cdcl<sub>NOT</sub>\*\* S T*  
**using** *inv rtrancp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtrancp-cdcl<sub>NOT</sub>-and-inv st n-d* **by** *blast*  
**have** *inv T*  
**using** *inv rtrancp-cdcl<sub>NOT</sub>-merged-bj-learn-inv st n-d* **by** *blast*  
**moreover**  
**have** *atms-of-mu (clauses T)  $\subseteq$  atms-of-m A* **and**  
*atm-of ' lits-of (trail T)  $\subseteq$  atms-of-m A*  
**using** *cdcl<sub>NOT</sub>.rtrancp-cdcl<sub>NOT</sub>-trail-clauses-bound[OF st'] inv atms-clss-S atms-trail-S n-d*  
**by** *blast+*  
**moreover moreover have** *no-dup (trail T)*  
**using** *cdcl<sub>NOT</sub>.rtrancp-cdcl<sub>NOT</sub>-no-dup[OF <cdcl<sub>NOT</sub>\*\* S T> inv n-d]* **by** *fast*  
**ultimately have** *set-mset (clauses U)*  
 $\subseteq$  *set-mset (not-simplified-cls (clauses T))  $\cup$  build-all-simple-clss (atms-of-m A)*  
**using** *cdcl<sub>NOT</sub> finite cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound*  
**by** (auto intro!: *cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound*)  
**moreover have** *set-mset (not-simplified-cls (clauses T))*  
 $\subseteq$  *set-mset (not-simplified-cls (clauses S))*  
**using** *rtrancp-cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing[OF st]* **by** auto  
**ultimately show** ?*case* **using** *IH inv atms-clss-S*  
**by** (auto dest!: *build-all-simple-clss-or-not-simplified-cls*)

**qed**

**abbreviation**  $\mu_{CDCL}'$ -*bound* **where**

$\mu_{CDCL}'$ -*bound A T* == ((2+card (atms-of-m A))  $\wedge$  (1+card (atms-of-m A))) \* 2  
+ card (set-mset (not-simplified-cls (clauses T)))  
+ 3  $\wedge$  card (atms-of-m A)

**lemma** *rtrancpl-cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound-card*:

**assumes**

*cdcl<sub>NOT</sub>-merged-bj-learn*\*\* *S T* **and**

*inv S* **and**

*atms-of-mu* (*clauses S*)  $\subseteq$  *atms-of-m A* **and**

*atm-of* ‘(*lits-of* (*trail S*))  $\subseteq$  *atms-of-m A* **and**

*n-d*: *no-dup* (*trail S*) **and**

*finite*: *finite A*

**shows**  $\mu_{CDCL}'\text{-merged } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$

**proof** –

**have** *set-mset* (*clauses T*)  $\subseteq$  *set-mset* (*not-simplified-cls*(*clauses S*))

$\cup$  *build-all-simple-clss* (*atms-of-m A*)

**using** *rtrancpl-cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound*[*OF assms*] .

**moreover have** *card* (*set-mset* (*not-simplified-cls*(*clauses S*))

$\cup$  *build-all-simple-clss* (*atms-of-m A*))

$\leq$  *card* (*set-mset* (*not-simplified-cls*(*clauses S*))) + 3  $\wedge$  *card* (*atms-of-m A*)

**by** (*meson* *Nat.le-trans* *atms-of-m-finite* *build-all-simple-clss-card* *card-Un-le* *finite*  
*nat-add-left-cancel-le*)

**ultimately have** *card* (*set-mset* (*clauses T*))

$\leq$  *card* (*set-mset* (*not-simplified-cls*(*clauses S*))) + 3  $\wedge$  *card* (*atms-of-m A*)

**by** (*meson* *build-all-simple-clss-finite* *card-mono* *dual-order.trans* *finite-UnI* *finite-set-mset*)

**moreover have** ((2 + *card* (*atms-of-m A*))  $\wedge$  (1 + *card* (*atms-of-m A*)) –  $\mu_C' A \ T$ ) \* 2

$\leq$  (2 + *card* (*atms-of-m A*))  $\wedge$  (1 + *card* (*atms-of-m A*)) \* 2

**by** *auto*

**ultimately show** *?thesis unfolding*  $\mu_{CDCL}'\text{-merged-def}$  **by** *auto*

**qed**

**sublocale** *cdcl<sub>NOT</sub>-increasing-restarts-ops*  $\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} \ \square \ S$

*cdcl<sub>NOT</sub>-merged-bj-learn* *f*

$\lambda A \ S. \ \text{atms-of-mu} \ (\text{clauses } S) \subseteq \text{atms-of-m } A$

$\wedge \text{atm-of } \text{‘lits-of } (\text{trail } S) \subseteq \text{atms-of-m } A \wedge \text{finite } A$

$\mu_{CDCL}'\text{-merged}$

$\lambda S. \ \text{inv } S \wedge \text{no-dup } (\text{trail } S)$

$\mu_{CDCL}'\text{-bound}$

**apply** *unfold-locales*

**using** *unbounded apply simp*

**using** *f-ge-1 apply force*

**apply** (*blast dest!*: *cdcl<sub>NOT</sub>-merged-bj-learn-is-trancpl-cdcl<sub>NOT</sub>* *trancpl-into-rtrancpl*  
*cdcl<sub>NOT</sub>.rtrancpl-cdcl<sub>NOT</sub>-trail-clauses-bound* )

**apply** (*simp add*: *cdcl<sub>NOT</sub>-decreasing-measure'*)

**using** *rtrancpl-cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound-card* **apply** *blast*

**apply** (*drule* *rtrancpl-cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing*)

**apply** (*auto dest!*: *simp*: *card-mono set-mset-mono* )  $\square$

**apply** *simp*

**apply** *auto*  $\square$

**using** *cdcl<sub>NOT</sub>-merged-bj-learn-no-dup-inv* *cdcl-merged-inv* **apply** *blast*

**apply** (*auto simp*: *inv-restart*)  $\square$

**done**

**lemma** *cdcl<sub>NOT</sub>-restart- $\mu_{CDCL}'\text{-merged-le-}\mu_{CDCL}'\text{-bound}$* :

**assumes**

*cdcl<sub>NOT</sub>-restart* *T V*

*inv* (*fst T*) **and**

*no-dup* (*trail* (*fst T*)) **and**

*atms-of-mu* (*clauses* (*fst T*))  $\subseteq$  *atms-of-m A* **and**



$atm\text{-}of \text{ ' } lits\text{-}of (trail (fst T)) \subseteq atm\text{-}of\text{-}m A$  and  
 $finite A$   
**shows**  $\mu_{CDCL}'\text{-merged } A (fst V) \leq \mu_{CDCL}'\text{-bound } A (fst T)$   
**using** *assms*  
**proof** *induction*  
**case** (*restart-full*  $S T n$ )  
**show** *?case*  
**unfolding** *fst-conv*  
**apply** (*rule* *rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound-card*)  
**using** *restart-full unfolding full1-def* **by** (*force dest!: tranclp-into-rtranclp*)+  
**next**  
**case** (*restart-step*  $m S T n U$ ) **note**  $st = this(1)$  **and**  $U = this(3)$  **and**  $inv = this(4)$  **and**  
 $n\text{-}d = this(5)$  **and**  $atms\text{-}clss = this(6)$  **and**  $atms\text{-}trail = this(7)$  **and**  $finite = this(8)$   
**then have**  $st'$ : *cdcl<sub>NOT</sub>-merged-bj-learn*\*\*  $S T$   
**by** (*blast dest: relpowp-imp-rtranclp*)  
**then have**  $st''$ : *cdcl<sub>NOT</sub>*\*\*  $S T$   
**using**  $inv\ n\text{-}d$  **apply** – **by** (*rule* *rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtranclp-cdcl<sub>NOT</sub>*) *auto*  
**have**  $inv T$   
**apply** (*rule* *rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-inv*)  
**using**  $inv\ st'\ n\text{-}d$  **by** *auto*  
**then have**  $inv U$   
**using**  $U$  **by** (*auto simp: inv-restart*)  
**have**  $atms\text{-}of\text{-}\mu (clauses T) \subseteq atm\text{-}of\text{-}m A$   
**using** *cdcl<sub>NOT</sub>.rtranclp-cdcl<sub>NOT</sub>-trail-clauses-bound*[*OF st'*]  $inv\ atms\text{-}clss\ atms\text{-}trail\ n\text{-}d$   
**by** *simp*  
**then have**  $atms\text{-}of\text{-}\mu (clauses U) \subseteq atm\text{-}of\text{-}m A$   
**using**  $U$  **by** *simp*  
**have**  $not\text{-}simplified\text{-}cls (clauses U) \subseteq \# not\text{-}simplified\text{-}cls (clauses T)$   
**using**  $\langle U \sim reduce\text{-}trail\text{-}to_{NOT} [] T \rangle$  **by** *auto*  
**moreover have**  $not\text{-}simplified\text{-}cls (clauses T) \subseteq \# not\text{-}simplified\text{-}cls (clauses S)$   
**apply** (*rule* *rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing*)  
**using**  $\langle (cdcl_{NOT}\text{-merged-bj-learn} \sim m) S T \rangle$  **by** (*auto dest!: relpowp-imp-rtranclp*)  
**ultimately have**  $U\text{-}S: not\text{-}simplified\text{-}cls (clauses U) \subseteq \# not\text{-}simplified\text{-}cls (clauses S)$   
**by** *auto*  
  
**have** (*set-mset* ( $clauses U$ ))  
 $\subseteq set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}m A)$   
**apply** (*rule* *rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-clauses-bound*)  
**apply** *simp*  
**using**  $\langle inv U \rangle$  **apply** *simp*  
**using**  $\langle atms\text{-}of\text{-}\mu (clauses U) \subseteq atm\text{-}of\text{-}m A \rangle$  **apply** *simp*  
**using**  $U$  **apply** *simp*  
**using**  $U$  **apply** *simp*  
**using** *finite* **apply** *simp*  
**done**  
**then have**  $f1: card (set\text{-}mset (clauses U)) \leq card (set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U))$   
 $\cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}m A))$   
**by** (*meson build-all-simple-clss-finite card-mono finite-UnI finite-set-mset*)  
  
**moreover have**  $set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}m A)$   
 $\subseteq set\text{-}mset (not\text{-}simplified\text{-}cls (clauses S)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}m A)$   
**using**  $U\text{-}S$  **by** *auto*  
**then have**  $f2:$   
 $card (set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}m A))$   
 $\leq card (set\text{-}mset (not\text{-}simplified\text{-}cls (clauses S)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}m A))$

by (meson build-all-simple-clss-finite card-mono finite-UnI finite-set-mset)

**moreover have** card (set-mset (not-simplified-cls (clauses S))  
 ∪ build-all-simple-clss (atms-of-m A))  
 ≤ card (set-mset (not-simplified-cls (clauses S))) + card (build-all-simple-clss (atms-of-m A))  
 using card-Un-le by blast

**moreover have** card (build-all-simple-clss (atms-of-m A)) ≤ 3 ^ card (atms-of-m A)  
 using atms-of-m-finite build-all-simple-clss-card local.finite by blast

**ultimately have** card (set-mset (clauses U))  
 ≤ card (set-mset (not-simplified-cls (clauses S))) + 3 ^ card (atms-of-m A)  
 by linarith

**then show** ?case **unfolding**  $\mu_{CDCL}'$ -merged-def by auto

qed

**lemma** cdcl<sub>NOT</sub>-restart- $\mu_{CDCL}'$ -bound-le- $\mu_{CDCL}'$ -bound:  
**assumes**  
 cdcl<sub>NOT</sub>-restart T V **and**  
 no-dup (trail (fst T)) **and**  
 inv (fst T) **and**  
 fin: finite A  
**shows**  $\mu_{CDCL}'$ -bound A (fst V) ≤  $\mu_{CDCL}'$ -bound A (fst T)  
 using assms(1-3)

**proof induction**  
**case** (restart-full S T n)  
**have** not-simplified-cls (clauses T) ⊆# not-simplified-cls (clauses S)  
**apply** (rule rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing)  
**using** ⟨full1 cdcl<sub>NOT</sub>-merged-bj-learn S T⟩ **unfolding** full1-def  
**by** (auto dest: tranclp-into-rtranclp)

**then show** ?case by (auto simp: card-mono set-mset-mono)

**next**  
**case** (restart-step m S T n U) **note** st = this(1) **and** U = this(3) **and** n-d = this(4) **and** inv = this(5)

**then have** st': cdcl<sub>NOT</sub>-merged-bj-learn\*\* S T  
**by** (blast dest: relpowp-imp-rtranclp)

**then have** st'': cdcl<sub>NOT</sub>\*\* S T  
**using** inv n-d **apply** – **by** (rule rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-is-rtranclp-cdcl<sub>NOT</sub>) auto

**have** inv T  
**apply** (rule rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-inv)  
**using** inv st' n-d by auto

**then have** inv U  
**using** U by (auto simp: inv-restart)

**have** not-simplified-cls (clauses U) ⊆# not-simplified-cls (clauses T)  
**using** ⟨U ~ reduce-trail-to<sub>NOT</sub> [] T⟩ by auto

**moreover have** not-simplified-cls (clauses T) ⊆# not-simplified-cls (clauses S)  
**apply** (rule rtranclp-cdcl<sub>NOT</sub>-merged-bj-learn-not-simplified-decreasing)  
**using** ⟨(cdcl<sub>NOT</sub>-merged-bj-learn ~ m) S T⟩ by (auto dest!: relpowp-imp-rtranclp)

**ultimately have** U-S: not-simplified-cls (clauses U) ⊆# not-simplified-cls (clauses S)  
 by auto

**then show** ?case by (auto simp: card-mono set-mset-mono)

qed

**sublocale** cdcl<sub>NOT</sub>-increasing-restarts - - - - - f λS T. T ~ reduce-trail-to<sub>NOT</sub> [] S  
 λA S. atms-of-mu (clauses S) ⊆ atms-of-m A  
 ∧ atm-of ' lits-of (trail S) ⊆ atms-of-m A ∧ finite A

```

 $\mu_{CDCL}'$ -merged  $cdcl_{NOT}$ -merged-bj-learn
   $\lambda S. inv\ S \wedge no\_dup\ (trail\ S)$ 
 $\lambda A\ T. ((2 + card\ (atms\_of\_m\ A)) \wedge (1 + card\ (atms\_of\_m\ A))) * 2$ 
   $+ card\ (set\_mset\ (not\_simplified\_cls(c\lause s\ T)))$ 
   $+ 3 \wedge card\ (atms\_of\_m\ A)$ 
apply unfold-locales
  using  $cdcl_{NOT}$ -restart- $\mu_{CDCL}'$ -merged-le- $\mu_{CDCL}'$ -bound apply force
  using  $cdcl_{NOT}$ -restart- $\mu_{CDCL}'$ -bound-le- $\mu_{CDCL}'$ -bound by fastforce

lemma  $cdcl_{NOT}$ -restart-eq-sat-iff:
assumes
   $cdcl_{NOT}$ -restart  $S\ T$  and
   $no\_dup\ (trail\ (fst\ S))$ 
   $inv\ (fst\ S)$ 
shows  $I \models_{sextm} clauses\ (fst\ S) \longleftrightarrow I \models_{sextm} clauses\ (fst\ T)$ 
using assms
proof (induction rule:  $cdcl_{NOT}$ -restart.induct)
case (restart-full  $S\ T\ n$ )
then have  $cdcl_{NOT}$ -merged-bj-learn**  $S\ T$ 
  by (simp add: tranclp-into-rtranclp full1-def)
then show ?case
  using  $cdcl_{NOT}$ .rtranclp-cdclNOT-bj-sat-ext-iff restart-full.prem s(1,2)
  rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT by auto
next
case (restart-step  $m\ S\ T\ n\ U$ )
then have  $cdcl_{NOT}$ -merged-bj-learn**  $S\ T$ 
  by (auto simp: tranclp-into-rtranclp full1-def dest!: relpowp-imp-rtranclp)
then have  $I \models_{sextm} clauses\ S \longleftrightarrow I \models_{sextm} clauses\ T$ 
  using  $cdcl_{NOT}$ .rtranclp-cdclNOT-bj-sat-ext-iff restart-step.prem s(1,2)
  rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT by auto
moreover have  $I \models_{sextm} clauses\ T \longleftrightarrow I \models_{sextm} clauses\ U$ 
  using restart-step.hyps(3) by auto
ultimately show ?case by auto
qed

lemma rtranclp-cdclNOT-restart-eq-sat-iff:
assumes
   $cdcl_{NOT}$ -restart**  $S\ T$  and
   $inv: inv\ (fst\ S)$  and  $n-d: no\_dup(trail\ (fst\ S))$ 
shows  $I \models_{sextm} clauses\ (fst\ S) \longleftrightarrow I \models_{sextm} clauses\ (fst\ T)$ 
using assms(1)
proof (induction rule: rtranclp-induct)
case base
then show ?case by simp
next
case (step  $T\ U$ ) note  $st = this(1)$  and  $cdcl = this(2)$  and  $IH = this(3)$ 
have  $inv\ (fst\ T)$  and  $no\_dup\ (trail\ (fst\ T))$ 
  using rtranclp-cdclNOT-with-restart-cdclNOT-inv using  $st\ inv\ n-d$  by blast+
then have  $I \models_{sextm} clauses\ (fst\ T) \longleftrightarrow I \models_{sextm} clauses\ (fst\ U)$ 
  using  $cdcl_{NOT}$ -restart-eq-sat-iff  $cdcl$  by blast
then show ?case using  $IH$  by blast
qed

lemma  $cdcl_{NOT}$ -restart-all-decomposition-implies-m:
assumes

```

```

  cdclNOT-restart  $S$   $T$  and
  inv: inv (fst  $S$ ) and  $n$ -d: no-dup(trail (fst  $S$ )) and
  all-decomposition-implies-m (clauses (fst  $S$ ))
    (get-all-marked-decomposition (trail (fst  $S$ )))
  shows all-decomposition-implies-m (clauses (fst  $T$ ))
    (get-all-marked-decomposition (trail (fst  $T$ )))
  using assms
proof (induction)
  case (restart-full  $S$   $T$   $n$ ) note full = this(1) and inv = this(2) and  $n$ -d = this(3) and
    decomp = this(4)
  have st: cdclNOT-merged-bj-learn**  $S$   $T$  and
     $n$ -s: no-step cdclNOT-merged-bj-learn  $T$ 
    using full unfolding full1-def by (fast dest: tranclp-into-rtranclp)+
  have st': cdclNOT**  $S$   $T$ 
    using inv rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT-and-inv st  $n$ -d by auto
  have inv  $T$ 
    using rtranclp-cdclNOT-cdclNOT-inv[OF st] inv  $n$ -d by auto
  then show ?case
    using cdclNOT.rtranclp-cdclNOT-all-decomposition-implies[OF - -  $n$ -d decomp] st' inv by auto
next
  case (restart-step  $m$   $S$   $T$   $n$   $U$ ) note st = this(1) and  $U$  = this(3) and inv = this(4) and
     $n$ -d = this(5) and decomp = this(6)
  show ?case using  $U$  by auto
qed

```

**lemma** rtranclp-cdcl<sub>NOT</sub>-restart-all-decomposition-implies-m:

```

  assumes
    cdclNOT-restart**  $S$   $T$  and
    inv: inv (fst  $S$ ) and  $n$ -d: no-dup(trail (fst  $S$ )) and
    decomp: all-decomposition-implies-m (clauses (fst  $S$ ))
      (get-all-marked-decomposition (trail (fst  $S$ )))
  shows all-decomposition-implies-m (clauses (fst  $T$ ))
    (get-all-marked-decomposition (trail (fst  $T$ )))
  using assms
proof (induction)
  case base
  then show ?case using decomp by simp
next
  case (step  $T$   $U$ ) note st = this(1) and cdcl = this(2) and IH = this(3)[OF this(4-)] and
    inv = this(4) and  $n$ -d = this(5) and decomp = this(6)
  have inv (fst  $T$ ) and no-dup (trail (fst  $T$ ))
    using rtranclp-cdclNOT-with-restart-cdclNOT-inv using st inv  $n$ -d by blast+
  then show ?case
    using cdclNOT-restart-all-decomposition-implies-m[OF cdcl] IH by auto
qed

```

**lemma** full-cdcl<sub>NOT</sub>-restart-normal-form:

```

  assumes
    full: full cdclNOT-restart  $S$   $T$  and
    inv: inv (fst  $S$ ) and  $n$ -d: no-dup(trail (fst  $S$ )) and
    decomp: all-decomposition-implies-m (clauses (fst  $S$ ))
      (get-all-marked-decomposition (trail (fst  $S$ ))) and
    atms-cl: atms-of-mu (clauses (fst  $S$ ))  $\subseteq$  atms-of-m  $A$  and
    atms-trail: atm-of ' lits-of (trail (fst  $S$ ))  $\subseteq$  atms-of-m  $A$  and
    fin: finite  $A$ 

```

**shows** *unsatisfiable* (*set-mset* (*clauses* (*fst S*)))  
 $\vee$  *lits-of* (*trail* (*fst T*))  $\models_{\text{sextm}}$  *clauses* (*fst S*)  $\wedge$  *satisfiable* (*set-mset* (*clauses* (*fst S*)))  
**proof** –  
**have** *inv-T*: *inv* (*fst T*) **and** *n-d-T*: *no-dup* (*trail* (*fst T*))  
**using** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-cdcl<sub>NOT</sub>-inv* **using** *full inv n-d unfolding full-def* **by** *blast+*  
**moreover have**  
*atms-cls-T*: *atms-of-mu* (*clauses* (*fst T*))  $\subseteq$  *atms-of-m* *A* **and**  
*atms-trail-T*: *atm-of* ‘*lits-of* (*trail* (*fst T*))  $\subseteq$  *atms-of-m* *A*  
**using** *rtrancpl-cdcl<sub>NOT</sub>-with-restart-bound-inv*[*of S T A*] *full atms-cls atms-trail fin inv n-d*  
**unfolding full-def** **by** *blast+*  
**ultimately have** *no-step cdcl<sub>NOT</sub>-merged-bj-learn* (*fst T*)  
**apply** –  
**apply** (*rule no-step-cdcl<sub>NOT</sub>-restart-no-step-cdcl<sub>NOT</sub>*[*of - A*])  
**using** *full unfolding full-def* **apply** *simp*  
**apply** *simp*  
**using** *fin* **apply** *simp*  
**done**  
**moreover have** *all-decomposition-implies-m* (*clauses* (*fst T*))  
(*get-all-marked-decomposition* (*trail* (*fst T*)))  
**using** *rtrancpl-cdcl<sub>NOT</sub>-restart-all-decomposition-implies-m*[*of S T*] *inv n-d decomp*  
*full unfolding full-def* **by** *auto*  
**ultimately have** *unsatisfiable* (*set-mset* (*clauses* (*fst T*)))  
 $\vee$  *trail* (*fst T*)  $\models_{\text{asm}}$  *clauses* (*fst T*)  $\wedge$  *satisfiable* (*set-mset* (*clauses* (*fst T*)))  
**apply** –  
**apply** (*rule cdcl<sub>NOT</sub>-merged-bj-learn-final-state*)  
**using** *atms-cls-T atms-trail-T fin n-d-T fin inv-T* **by** *blast+*  
**then consider**  
(*unsat*) *unsatisfiable* (*set-mset* (*clauses* (*fst T*)))  
| (*sat*) *trail* (*fst T*)  $\models_{\text{asm}}$  *clauses* (*fst T*) **and** *satisfiable* (*set-mset* (*clauses* (*fst T*)))  
**by** *auto*  
**then show** *unsatisfiable* (*set-mset* (*clauses* (*fst S*)))  
 $\vee$  *lits-of* (*trail* (*fst T*))  $\models_{\text{sextm}}$  *clauses* (*fst S*)  $\wedge$  *satisfiable* (*set-mset* (*clauses* (*fst S*)))  
**proof** *cases*  
**case** *unsat*  
**then have** *unsatisfiable* (*set-mset* (*clauses* (*fst S*)))  
**unfolding** *satisfiable-def* **apply** *auto*  
**using** *rtrancpl-cdcl<sub>NOT</sub>-restart-eq-sat-iff*[*of S T*] *full inv n-d*  
*consistent-true-clss-ext-satisfiable true-clss-imp-true-clss-ext*  
**unfolding** *satisfiable-def full-def* **by** *blast*  
**then show** *?thesis* **by** *blast*  
**next**  
**case** *sat*  
**then have** *lits-of* (*trail* (*fst T*))  $\models_{\text{sextm}}$  *clauses* (*fst T*)  
**using** *true-clss-imp-true-clss-ext* **by** (*auto simp: true-annots-true-clss*)  
**then have** *lits-of* (*trail* (*fst T*))  $\models_{\text{sextm}}$  *clauses* (*fst S*)  
**using** *rtrancpl-cdcl<sub>NOT</sub>-restart-eq-sat-iff*[*of S T*] *full inv n-d* **unfolding** *full-def* **by** *blast*  
**moreover then have** *satisfiable* (*set-mset* (*clauses* (*fst S*)))  
**using** *consistent-true-clss-ext-satisfiable distinctconsistent-interp n-d-T* **by** *fast*  
**ultimately show** *?thesis* **by** *fast*  
**qed**  
**qed**

**corollary** *full-cdcl<sub>NOT</sub>-restart-normal-form-init-state*:

**assumes**

*init-state*: *trail S* = [] *clauses S* = *N* **and**

```

    full: full cdclNOT-restart (S, 0) T and
    inv: inv S
shows unsatisfiable (set-mset N)
    ∨ lits-of (trail (fst T)) ⊨sextm N ∧ satisfiable (set-mset N)
using full-cdclNOT-restart-normal-form[of (S, 0) T] assms by auto

end

end
theory DPLL-NOT
imports CDCL-NOT
begin

```

## 15 DPLL as an instance of NOT

### 15.1 DPLL with simple backtrack

```

locale dpll-with-backtrack
begin
inductive backtrack :: ('v, unit, unit) marked-lit list × 'v clauses
  ⇒ ('v, unit, unit) marked-lit list × 'v clauses ⇒ bool where
  backtrack-split (fst S) = (M', L # M) ⇒ is-marked L ⇒ D ∈# snd S
  ⇒ fst S ⊨as CNot D ⇒ backtrack S (Propagated (− (lit-of L)) () # M, snd S)

inductive-cases backtrackE[elim]: backtrack (M, N) (M', N')
lemma backtrack-is-backjump:
  fixes M M' :: ('v, unit, unit) marked-lit list
  assumes
    backtrack: backtrack (M, N) (M', N') and
    no-dup: (no-dup ∘ fst) (M, N) and
    decomp: all-decomposition-implies-m N (get-all-marked-decomposition M)
  shows
    ∃ C F' K F L l C'.
      M = F' @ Marked K () # F ∧
      M' = Propagated L l # F ∧ N = N' ∧ C ∈# N ∧ F' @ Marked K d # F ⊨as CNot C ∧
      undefined-lit F L ∧ atm-of L ∈ atms-of-mu N ∪ atm-of ' lits-of (F' @ Marked K d # F) ∧
      N ⊨pm C' + {#L#} ∧ F ⊨as CNot C'

proof −
  let ?S = (M, N)
  let ?T = (M', N')
  obtain F F' P L D where
    b-sp: backtrack-split M = (F', L # F) and
    is-marked L and
    D ∈# snd ?S and
    M ⊨as CNot D and
    bt: backtrack ?S (Propagated (− (lit-of L)) P # F, N) and
    M': M' = Propagated (− (lit-of L)) P # F and
    [simp]: N' = N
  using backtrackE[OF backtrack] by (metis backtrack fstI sndI)
  let ?K = lit-of L
  let ?C = image-mset lit-of {#K ∈# mset M. is-marked K ∧ K ≠ L#} :: 'v literal multiset
  let ?C' = set-mset (image-mset single (?C + {#?K#}))
  obtain K where L: L = Marked K () using ⟨is-marked L⟩ by (cases L) auto

  have M: M = F' @ Marked K () # F

```

```

  using b-sp by (metis L backtrack-split-list-eq fst-conv snd-conv)
moreover have  $F' @ \text{Marked } K () \# F \models_{as} C \text{Not } D$ 
  using  $\langle M \models_{as} C \text{Not } D \rangle$  unfolding M .
moreover have undefined-lit F ( $- ?K$ )
  using no-dup unfolding M L by (simp add: defined-lit-map)
moreover have  $\text{atm-of } (-K) \in \text{atms-of-mu } N \cup \text{atm-of ' lits-of } (F' @ \text{Marked } K d \# F)$ 
  by auto
moreover
  have  $\text{set-mset } N \cup ?C' \models_{ps} \{\{\#\}\}$ 
  proof -
    have A:  $\text{set-mset } N \cup ?C' \cup (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M =$ 
       $\text{set-mset } N \cup (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M$ 
    unfolding M L by auto
    have  $\text{set-mset } N \cup \{\{\#\text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
       $\models_{ps} (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
    moreover have  $C': ?C' = \{\{\#\text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
    unfolding M L apply standard
    apply force
    using IntI by auto
    ultimately have N-C-M:  $\text{set-mset } N \cup ?C' \models_{ps} (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M$ 
    by auto
    have  $\text{set-mset } N \cup (\lambda L. \{\#\text{lit-of } L\# \}) \text{ ' (set } M) \models_{ps} \{\{\#\}\}$ 
    unfolding true-clss-clss-def
    proof (intro allI impI, goal-cases)
      case (1 I) note tot = this(1) and cons = this(2) and I-N-M = this(3)
      have I  $\models D$ 
      using I-N-M  $\langle D \in \# \text{ snd } ?S \rangle$  unfolding true-clss-def by auto
      moreover have I  $\models_s C \text{Not } D$ 
      using  $\langle M \models_{as} C \text{Not } D \rangle$  unfolding M by (metis 1(3)  $\langle M \models_{as} C \text{Not } D \rangle$ 
        true-annots-true-clss true-clss-mono-set-mset-l true-clss-def
        true-clss-singleton-lit-of-implies-incl true-clss-union)
      ultimately show ?case using cons consistent-CNot-not by blast
    qed
  then show ?thesis
    using true-clss-clss-left-right[OF N-C-M, of  $\{\{\#\}\}$ ] unfolding A by auto
  qed
have  $N \models_{pm} \text{image-mset } \text{uminus } ?C + \{\# - ?K\# \}$ 
unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
proof (intro allI impI)
  fix I
  assume
    tot:  $\text{total-over-set } I (\text{atms-of-m } (\text{set-mset } N \cup \{\text{image-mset } \text{uminus } ?C + \{\# - ?K\# \}))$  and
    cons:  $\text{consistent-interp } I$  and
     $I \models_{sm} N$ 
  have  $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$ 
  using cons tot unfolding consistent-interp-def L by (cases K) auto
  have  $\text{total-over-set } I (\text{atm-of ' lit-of ' (set } M \cap \{L. \text{is-marked } L \wedge L \neq \text{Marked } K d\}))$ 
  using tot by (auto simp add: L atms-of-uminus-lit-atm-of-lit-of)

then have H:  $\bigwedge x.$ 
   $\text{lit-of } x \notin I \implies x \in \text{set } M \implies \text{is-marked } x$ 
   $\implies x \neq \text{Marked } K d \implies -\text{lit-of } x \in I$ 

unfolding total-over-set-def atms-of-s-def

```

```

proof –
  fix  $x :: ('v, unit, unit) \text{ marked-lit}$ 
  assume  $a1: x \in \text{set } M$ 
  assume  $a2: \forall l \in \text{atm-of } ' \text{ lit-of } ' (\text{set } M \cap \{L. \text{ is-marked } L \wedge L \neq \text{Marked } K \ d\}).$ 
     $\text{Pos } l \in I \vee \text{Neg } l \in I$ 
  assume  $a3: \text{lit-of } x \notin I$ 
  assume  $a4: \text{is-marked } x$ 
  assume  $a5: x \neq \text{Marked } K \ d$ 
  have  $f6: \text{Neg } (\text{atm-of } (\text{lit-of } x)) = - \text{Pos } (\text{atm-of } (\text{lit-of } x))$ 
    by simp
  have  $\text{Pos } (\text{atm-of } (\text{lit-of } x)) \in I \vee \text{Neg } (\text{atm-of } (\text{lit-of } x)) \in I$ 
    using  $a5 \ a4 \ a2 \ a1$  by blast
  then show  $-\text{lit-of } x \in I$ 
    using  $f6 \ a3$  by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1))
  qed
have  $\neg I \models_s ?C'$ 
  using  $\langle \text{set-mset } N \cup ?C' \models_{ps} \{\{\#\}\} \rangle \text{ tot cons } \langle I \models_{sm} N \rangle$ 
  unfolding true-clss-clss-def total-over-m-def
  by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-m-single-image-atm-of-lit-of)
  then show  $I \models \text{image-mset } \text{uminus } ?C + \{\#\text{-lit-of } L\#\}$ 
  unfolding true-clss-def true-clss-def Bex-mset-def
  using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
  unfolding  $L$  by (auto dest!: H)
  qed
moreover
  have  $\text{set } F' \cap \{K. \text{ is-marked } K \wedge K \neq L\} = \{\}$ 
    using backtrack-split-fst-not-marked[of - M] b-sp by auto
  then have  $F \models_{as} \text{CNot } (\text{image-mset } \text{uminus } ?C)$ 
    unfolding  $M \text{ CNot-def true-annots-def}$  by (auto simp add: L lits-of-def)
  ultimately show ?thesis
    using  $M' \langle D \in \# \text{ snd } ?S \rangle L$  by force
qed

lemma backtrack-is-backjump':
  fixes  $M \ M' :: ('v, unit, unit) \text{ marked-lit list}$ 
  assumes
    backtrack: backtrack S T and
    no-dup: (no-dup  $\circ$  fst) S and
    decomp: all-decomposition-implies-m (snd S) (get-all-marked-decomposition (fst S))
  shows
     $\exists C \ F' \ K \ F \ L \ l \ C'.$ 
     $\text{fst } S = F' @ \text{Marked } K \ () \ \# \ F \wedge$ 
     $T = (\text{Propagated } L \ l \ \# \ F, \text{snd } S) \wedge C \in \# \text{ snd } S \wedge \text{fst } S \models_{as} \text{CNot } C$ 
     $\wedge \text{undefined-lit } F \ L \wedge \text{atm-of } L \in \text{atms-of-mu } (\text{snd } S) \cup \text{atm-of } ' \text{ lits-of } (\text{fst } S) \wedge$ 
     $\text{snd } S \models_{pm} C' + \{\#L\#\} \wedge F \models_{as} \text{CNot } C'$ 
  apply (cases S, cases T)
  using backtrack-is-backjump[of fst S snd S fst T snd T] assms by fastforce

sublocale dpll-state fst snd  $\lambda L \ (M, N). (L \ \# \ M, N) \ \lambda(M, N). (tl \ M, N)$ 
   $\lambda C \ (M, N). (M, \{\#C\#\} + N) \ \lambda C \ (M, N). (M, \text{remove-mset } C \ N)$ 
  by unfold-locales auto

sublocale backjumping-ops fst snd  $\lambda L \ (M, N). (L \ \# \ M, N) \ \lambda(M, N). (tl \ M, N)$ 
   $\lambda C \ (M, N). (M, \{\#C\#\} + N) \ \lambda C \ (M, N). (M, \text{remove-mset } C \ N) \ \lambda - \ S \ T. \text{backtrack } S \ T$ 

```



by *unfold-locales*

**lemma** *backtrack-is-backjump''*:

**fixes**  $M M' :: ('v, unit, unit) \text{ marked-lit list}$

**assumes**

*backtrack*: *backtrack*  $S T$  **and**

*no-dup*:  $(no\text{-}dup \circ fst) S$  **and**

*decomp*: *all-decomposition-implies-m*  $(snd S)$  (*get-all-marked-decomposition*  $(fst S)$ )

**shows** *backjump*  $S T$

**proof** –

**obtain**  $C F' K F L l C'$  **where**

1:  $fst S = F' @ \text{Marked } K () \# F$  **and**

2:  $T = (\text{Propagated } L l \# F, snd S)$  **and**

3:  $C \in \# snd S$  **and**

4:  $fst S \models_{as} C \text{Not } C$  **and**

5: *undefined-lit*  $F L$  **and**

6:  $atm\text{-}of L \in atm\text{-}of\text{-}\mu (snd S) \cup atm\text{-}of ' \text{ lits-of } (fst S)$  **and**

7:  $snd S \models_{pm} C' + \{\#L\# \}$  **and**

8:  $F \models_{as} C \text{Not } C'$

**using** *backtrack-is-backjump'*[*OF* *assms*] **by** *blast*

**show** *?thesis*

**using** *backjump.intros*[*OF* 1 - 3 4 5 6 7 8] 2 *backtrack* 1 5

**by** (*auto simp: state-eq<sub>NOT</sub>-def simp del: state-simp<sub>NOT</sub>*)

**qed**

**lemma** *can-do-bt-step*:

**assumes**

$M: fst S = F' @ \text{Marked } K d \# F$  **and**

$C \in \# snd S$  **and**

$C: fst S \models_{as} C \text{Not } C$

**shows**  $\neg no\text{-}step \text{ backtrack } S$

**proof** –

**obtain**  $L G' G$  **where**

*backtrack-split*  $(fst S) = (G', L \# G)$

**unfolding**  $M$  **by** (*induction*  $F'$  *rule: marked-lit-list-induct*) *auto*

**moreover then have** *is-marked*  $L$

**by** (*metis backtrack-split-snd-hd-marked list.distinct*(1) *list.sel*(1) *snd-conv*)

**ultimately show** *?thesis*

**using** *backtrack.intros*[*of*  $S G' L G C$ ]  $\langle C \in \# snd S \rangle C$  **unfolding**  $M$  **by** *auto*

**qed**

**end**

**sublocale** *dpll-with-backtrack*  $\subseteq$  *dpll-with-backjumping-ops* *fst snd*  $\lambda L (M, N). (L \# M, N)$

$\lambda (M, N). (tl M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, \text{remove-mset } C N) \lambda - -. \text{True}$

$\lambda (M, N). no\text{-}dup M \wedge all\text{-}decomposition\text{-}implies\text{-}m N (\text{get-all-marked-decomposition } M)$

$(\lambda - - S T. \text{backtrack } S T)$

**by** *unfold-locales* (*metis* (*mono-tags, lifting*) *dpll-with-backtrack.backtrack-is-backjump''*

*dpll-with-backtrack.can-do-bt-step prod.case-eq-if comp-apply*)

**sublocale** *dpll-with-backtrack*  $\subseteq$  *dpll-with-backjumping* *fst snd*  $\lambda L (M, N). (L \# M, N)$

$\lambda (M, N). (tl M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, \text{remove-mset } C N) \lambda - -. \text{True}$

$\lambda (M, N). no\text{-}dup M \wedge all\text{-}decomposition\text{-}implies\text{-}m N (\text{get-all-marked-decomposition } M)$

$(\lambda - - S T. \text{backtrack } S T)$

**apply** *unfold-locales*

**using** *dpll-bj-no-dup dpll-bj-all-decomposition-implies-inv* **apply** *fastforce*  
**done**

**sublocale** *dpll-with-backtrack*  $\subseteq$  *conflict-driven-clause-learning-ops*  
*fst snd*  $\lambda L (M, N). (L \# M, N)$   
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, remove\_mset\ C\ N) \lambda - -. True$   
 $\lambda(M, N). no\_dup\ M \wedge all\_decomposition\_implies\_m\ N\ (get\_all\_marked\_decomposition\ M)$   
 $(\lambda - -\ S\ T. backtrack\ S\ T) \lambda - -. False\ \lambda - -. False$   
**by** *unfold-locales*

**sublocale** *dpll-with-backtrack*  $\subseteq$  *conflict-driven-clause-learning*  
*fst snd*  $\lambda L (M, N). (L \# M, N)$   
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, remove\_mset\ C\ N) \lambda - -. True$   
 $\lambda(M, N). no\_dup\ M \wedge all\_decomposition\_implies\_m\ N\ (get\_all\_marked\_decomposition\ M)$   
 $(\lambda - -\ S\ T. backtrack\ S\ T) \lambda - -. False\ \lambda - -. False$   
**apply** *unfold-locales*  
**using** *cdcl<sub>NOT</sub>.simps dpll-bj-inv forgetE learnE* **by** *blast*

**context** *dpll-with-backtrack*

**begin**

**lemma** *wf-tranclp-dpll-initail-state:*

**assumes** *fin: finite A*

**shows** *wf*  $\{((M'::('v, unit, unit)\ marked\_lits, N'::'v\ clauses), ([], N)) | M'\ N'\ N.$

$dpll\_bj^{++}\ ([], N)\ (M', N') \wedge atms\_of\_mu\ N \subseteq atms\_of\_m\ A\}$

**using** *wf-tranclp-dpll-bj[OF assms(1)]* **by** *(rule wf-subset) auto*

**corollary** *full-dpll-final-state-conclusive:*

**fixes** *M M' :: ('v, unit, unit) marked-lit list*

**assumes**

*full: full dpll-bj ([], N) (M', N')*

**shows** *unsatisfiable (set-mset N)  $\vee$  (M'  $\models_{asm}$  N  $\wedge$  satisfiable (set-mset N))*

**using** *assms full-dpll-backjump-final-state[of ([],N) (M', N') set-mset N]* **by** *auto*

**corollary** *full-dpll-normal-form-from-init-state:*

**fixes** *M M' :: ('v, unit, unit) marked-lit list*

**assumes**

*full: full dpll-bj ([], N) (M', N')*

**shows** *M'  $\models_{asm}$  N  $\longleftrightarrow$  satisfiable (set-mset N)*

**proof** –

**have** *no-dup M'*

**using** *rtranclp-dpll-bj-no-dup[of ([], N) (M', N')]*

*full* **unfolding** *full-def* **by** *auto*

**then have** *M'  $\models_{asm}$  N  $\implies$  satisfiable (set-mset N)*

**using** *distinctconsistent-interp satisfiable-carac' true-annots-true-cls* **by** *blast*

**then show** *?thesis*

**using** *full-dpll-final-state-conclusive[OF full]* **by** *auto*

**qed**

**lemma** *cdcl<sub>NOT</sub>-is-dpll:*

*cdcl<sub>NOT</sub> S T  $\longleftrightarrow$  dpll-bj S T*

**by** *(auto simp: cdcl<sub>NOT</sub>.simps learn.simps forget<sub>NOT</sub>.simps)*

Another proof of termination:

**lemma** *wf*  $\{(T, S). dpll\_bj\ S\ T \wedge cdcl_{NOT}\ NOT\_all\_inv\ A\ S\}$

**unfolding** *cdcl<sub>NOT</sub>-is-dpll[symmetric]*

```

by (rule wf-cdclNOT-no-learn-and-forget-infinite-chain)
(auto simp: learn.simps forgetNOT.simps)
end

```

## 15.2 Adding restarts

```

locale dpll-withbacktrack-and-restarts =
  dpll-with-backtrack +
  fixes f :: nat  $\Rightarrow$  nat
  assumes unbounded: unbounded f and f-ge-1:  $\bigwedge n. n \geq 1 \implies f\ n \geq 1$ 
begin
  sublocale cdclNOT-increasing-restarts fst snd  $\lambda L\ (M, N). (L \# M, N) \lambda(M, N). (tl\ M, N)$ 
     $\lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, \text{remove-mset } C\ N) f \lambda(-, N) S. S = ([], N)$ 
     $\lambda A\ (M, N). \text{atms-of-mu } N \subseteq \text{atms-of-m } A \wedge \text{atm-of } \text{'lits-of } M \subseteq \text{atms-of-m } A \wedge \text{finite } A$ 
     $\wedge \text{all-decomposition-implies-m } N\ (\text{get-all-marked-decomposition } M)$ 
     $\lambda A\ T. (2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))$ 
     $- \mu_C (1 + \text{card } (\text{atms-of-m } A)) (2 + \text{card } (\text{atms-of-m } A)) (\text{trail-weight } T) \text{ dpll-bj}$ 
     $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N\ (\text{get-all-marked-decomposition } M)$ 
     $\lambda A\ -. (2 + \text{card } (\text{atms-of-m } A)) \wedge (1 + \text{card } (\text{atms-of-m } A))$ 
  apply unfold-locales
    apply (rule unbounded)
    using f-ge-1 apply fastforce
    apply (smt dpll-bj-all-decomposition-implies-inv dpll-bj-atms-in-trail-in-set
      dpll-bj-clauses dpll-bj-no-dup prod.case-eq-if)
    apply (rule dpll-bj-trail-mes-decreasing-prop; auto)
    apply (case-tac T, simp)
    apply (case-tac U, simp)
    using dpll-bj-clauses dpll-bj-all-decomposition-implies-inv dpll-bj-no-dup by fastforce+
end

end
theory DPLL-W
imports Main Partial-Clausal-Logic Partial-Annotated-Clausal-Logic List-More Wellfounded-More
  DPLL-NOT
begin

```

## 16 DPLL

### 16.1 Rules

```

type-synonym 'a dpllW-marked-lit = ('a, unit, unit) marked-lit
type-synonym 'a dpllW-marked-lits = ('a, unit, unit) marked-lits
type-synonym 'v dpllW-state = 'v dpllW-marked-lits  $\times$  'v clauses

```

```

abbreviation trail :: 'v dpllW-state  $\Rightarrow$  'v dpllW-marked-lits where
  trail  $\equiv$  fst
abbreviation clauses :: 'v dpllW-state  $\Rightarrow$  'v clauses where
  clauses  $\equiv$  snd

```

The definition of DPLL is given in figure 2.13 page 70 of CW.

```

inductive dpllW :: 'v dpllW-state  $\Rightarrow$  'v dpllW-state  $\Rightarrow$  bool where
  propagate:  $C + \{\#L\# \} \in \# \text{ clauses } S \implies \text{trail } S \models_{\text{as}} C \text{Not } C \implies \text{undefined-lit } (\text{trail } S) L$ 
     $\implies \text{dpll}_W S (\text{Propagated } L\ () \# \text{trail } S, \text{clauses } S) |$ 
  decided:  $\text{undefined-lit } (\text{trail } S) L \implies \text{atm-of } L \in \text{atms-of-mu } (\text{clauses } S)$ 
     $\implies \text{dpll}_W S (\text{Marked } L\ () \# \text{trail } S, \text{clauses } S) |$ 

```

*backtrack*:  $\text{backtrack-split } (\text{trail } S) = (M', L \# M) \implies \text{is-marked } L \implies D \in \# \text{ clauses } S$   
 $\implies \text{trail } S \models_{\text{as}} C \text{Not } D \implies \text{dpll}_W S (\text{Propagated } (- (\text{lit-of } L)) () \# M, \text{ clauses } S)$

## 16.2 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')*  
**using** *assms*

**proof** (*induct rule: dpll<sub>W</sub>.induct*)

**case** (*decided L S*)

**then show** *?case* **using** *defined-lit-map* **by force**

**next**

**case** (*propagate C L S*)

**then show** *?case* **using** *defined-lit-map* **by force**

**next**

**case** (*backtrack S M' L M D*) **note** *extracted = this(1)* **and** *no-dup = this(5)*

**show** *?case*

**using** *no-dup backtrack-split-list-eq[of trail S, symmetric]* **unfolding** *extracted* **by auto**

**qed**

**lemma** *dpll<sub>W</sub>-consistent-interp-inv*:

**assumes** *dpll<sub>W</sub> S S'*

**and** *consistent-interp (lits-of (trail S))*

**and** *no-dup (trail S)*

**shows** *consistent-interp (lits-of (trail S'))*

**using** *assms*

**proof** (*induct rule: dpll<sub>W</sub>.induct*)

**case** (*backtrack S M' L M D*) **note** *extracted = this(1)* **and** *marked = this(2)* **and** *D = this(4)* **and**  
*cons = this(5)* **and** *no-dup = this(6)*

**have** *no-dup'*: *no-dup M*

**by** (*metis (no-types) backtrack-split-list-eq distinct.simps(2) distinct-append extracted*  
*list.simps(9) map-append no-dup snd-conv*)

**then have** *insert (lit-of L) (lits-of M) ⊆ lits-of (trail S)*

**using** *backtrack-split-list-eq[of trail S, symmetric]* **unfolding** *extracted* **by auto**

**then have** *cons*: *consistent-interp (insert (lit-of L) (lits-of M))*

**using** *consistent-interp-subset cons* **by blast**

**moreover**

**have** *lit-of L ∉ lits-of M*

**using** *no-dup backtrack-split-list-eq[of trail S, symmetric]* *extracted*  
**unfolding** *lits-of-def* **by force**

**moreover**

**have** *atm-of (−lit-of L) ∉ (λm. atm-of (lit-of m)) ‘ set M*

**using** *no-dup backtrack-split-list-eq[of trail S, symmetric]* **unfolding** *extracted* **by force**

**then have** *−lit-of L ∉ lits-of M*

**unfolding** *lits-of-def* **by force**

**ultimately show** *?case* **by simp**

**qed** (*auto intro: consistent-add-undefined-lit-consistent*)

**lemma** *dpll<sub>W</sub>-vars-in-snd-inv*:

**assumes** *dpll<sub>W</sub> S S'*

**and** *atm-of ‘ (lits-of (trail S)) ⊆ atms-of-mu (clauses S)*

**shows** *atm-of ‘ (lits-of (trail S')) ⊆ atms-of-mu (clauses S')*

**using** *assms*

**proof** (*induct rule: dpll<sub>W</sub>.induct*)

```

case (backtrack S M' L M D)
then have atm-of (lit-of L) ∈ atms-of-mu (clauses S)
  using backtrack-split-list-eq[of trail S, symmetric] by auto
moreover
  have atm-of ' lits-of (trail S) ⊆ atms-of-mu (clauses S)
    using backtrack(5) by simp
  then have ∧xb. xb ∈ set M ⇒ atm-of (lit-of xb) ∈ atms-of-mu (clauses S)
    using backtrack-split-list-eq[symmetric, of trail S] backtrack.hyps(1)
    unfolding lits-of-def by auto
  ultimately show ?case by (auto simp : lits-of-def)
qed (auto simp: in-plus-implies-atm-of-on-atms-of-m)

```

**lemma** *atms-of-m-lit-of-atms-of*: *atms-of-m* ((λa. {#lit-of a#}) ' c) = *atm-of* ' *lit-of* ' c  
**unfolding** *atms-of-m-def* **using** *image-iff* **by** *force*

Lemma theorem 2.8.2 page 71 of CW

**lemma** *dpll<sub>W</sub>-propagate-is-conclusion*:

```

assumes dpllW S S'
and all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
and atm-of ' lits-of (trail S) ⊆ atms-of-mu (clauses S)
shows all-decomposition-implies-m (clauses S') (get-all-marked-decomposition (trail S'))
using assms

```

**proof** (*induct rule: dpll<sub>W</sub>.induct*)

**case** (*decided L S*)

**then show** ?case **unfolding** *all-decomposition-implies-def* **by** *simp*

**next**

**case** (*propagate C L S*) **note** *inS* = *this*(1) **and** *cnot* = *this*(2) **and** *IH* = *this*(4) **and** *undef* = *this*(3) **and** *atms-incl* = *this*(5)

**let** ?*I* = *set* (map (λa. {#lit-of a#}) (trail S)) ∪ *set-mset* (clauses S)

**have** ?*I* ⊢<sub>p</sub> C + {#L#} **by** (auto simp add: *inS*)

**moreover have** ?*I* ⊢<sub>ps</sub> CNot C **using** *true-annots-true-clss-cls cnot* **by** *fastforce*

**ultimately have** ?*I* ⊢<sub>p</sub> {#L#} **using** *true-clss-cls-plus-CNot*[of ?*I* C L] *inS* **by** *blast*

```

{
  assume get-all-marked-decomposition (trail S) = []
  then have ?case by blast
}

```

**moreover** {

**assume** *n*: *get-all-marked-decomposition* (trail S) ≠ []

**have** 1: ∧a b. (a, b) ∈ *set* (*tl* (*get-all-marked-decomposition* (trail S)))

⇒ ((λa. {#lit-of a#}) ' *set* a ∪ *set-mset* (clauses S)) ⊢<sub>ps</sub> (λa. {#lit-of a#}) ' *set* b  
**using** *IH* **unfolding** *all-decomposition-implies-def* **by** (*fastforce simp add: list.set-sel*(2) *n*)

**moreover have** 2: ∧a c. *hd* (*get-all-marked-decomposition* (trail S)) = (a, c)

⇒ ((λa. {#lit-of a#}) ' *set* a ∪ *set-mset* (clauses S)) ⊢<sub>ps</sub> ((λa. {#lit-of a#}) ' *set* c)  
**by** (*metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single list.collapse n*)

**moreover have** 3: ∧a c. *hd* (*get-all-marked-decomposition* (trail S)) = (a, c)

⇒ ((λa. {#lit-of a#}) ' *set* a ∪ *set-mset* (clauses S)) ⊢<sub>p</sub> {#L#}

**proof** –

**fix** a c

**assume** *h*: *hd* (*get-all-marked-decomposition* (trail S)) = (a, c)

**have** *h'*: trail S = c @ a **using** *get-all-marked-decomposition-decomp h* **by** *blast*

**have** *I*: *set* (map (λa. {#lit-of a#}) a) ∪ *set-mset* (clauses S)

∪ (λa. {#lit-of a#}) ' *set* c ⊢<sub>ps</sub> CNot C

**using** ( ?*I* ⊢<sub>ps</sub> CNot C ) **unfolding** *h'* **by** (*simp add: Un-commute Un-left-commute*)  
**have**

```

  atms-of-m (CNot C) ⊆ atms-of-m (set (map (λa. {#lit-of a#}) a) ∪ set-mset (clauses S))
  and
  atms-of-m ((λa. {#lit-of a#}) ‘ set c) ⊆ atms-of-m (set (map (λa. {#lit-of a#}) a)
    ∪ set-mset (clauses S))
  apply (metis CNot-plus Un-subset-iff atms-of-atms-of-m-mono atms-of-m-CNot-atms-of
    atms-of-m-union inS mem-set-mset-iff sup.coboundedI2)
  using inS atms-of-atms-of-m-mono atms-incl by (fastforce simp: h')

  then have (λa. {#lit-of a#}) ‘ set a ∪ set-mset (clauses S) ⊨ps CNot C
    using true-clss-clss-left-right[OF - I] h 2 by auto
  then show (λa. {#lit-of a#}) ‘ set a ∪ set-mset (clauses S) ⊨p {#L#}
    by (metis (no-types) Un-insert-right inS insertI1 mk-disjoint-insert inS mem-set-mset-iff
      true-clss-clss-in true-clss-clss-plus-CNot)
  qed
  ultimately have ?case
    by (case-tac hd (get-all-marked-decomposition (trail S)))
      (auto simp add: all-decomposition-implies-def)
}
ultimately show ?case by auto
next
case (backtrack S M' L M D) note extracted = this(1) and marked = this(2) and D = this(3) and
  cnot = this(4) and cons = this(4) and IH = this(5) and atms-incl = this(6)
have S: trail S = M' @ L # M
  using backtrack-split-list-eq[of trail S] unfolding extracted by auto
have M': ∀ l ∈ set M'. ¬is-marked l
  using extracted backtrack-split-fst-not-marked[of - trail S] by simp
have n: get-all-marked-decomposition (trail S) ≠ [] by auto
then have all-decomposition-implies-m (clauses S) ((L # M, M')
  # tl (get-all-marked-decomposition (trail S)))
  by (metis (no-types) IH extracted get-all-marked-decomposition-backtrack-split list.exhaust-sel)
then have 1: (λa. {#lit-of a#}) ‘ set (L # M) ∪ set-mset (clauses S) ⊨ps(λa. {#lit-of a#}) ‘ set
M'
  by simp
moreover
have (λa. {#lit-of a#}) ‘ set (L # M) ∪ (λa. {#lit-of a#}) ‘ set M' ⊨ps CNot D
  by (metis (mono-tags, lifting) S Un-commute cons image-Un set-append
    true-annots-true-clss-clss)
then have 2: (λa. {#lit-of a#}) ‘ set (L # M) ∪ set-mset (clauses S) ∪ (λa. {#lit-of a#}) ‘ set
M'
  ⊨ps CNot D
  by (metis (no-types, lifting) Un-assoc Un-left-commute true-clss-clss-union-l-r)
ultimately
have set (map (λa. {#lit-of a#}) (L # M)) ∪ set-mset (clauses S) ⊨ps CNot D
  using true-clss-clss-left-right by fastforce
then have set (map (λa. {#lit-of a#}) (L # M)) ∪ set-mset (clauses S) ⊨p {#}
  by (metis (mono-tags, lifting) D Un-def mem-Collect-eq set-mset-def
    true-clss-clss-contradiction-true-clss-clss-false)
then have IL: (λa. {#lit-of a#}) ‘ set M ∪ set-mset (clauses S) ⊨p {#-lit-of L#}
  using true-clss-clss-false-left-right by auto
show ?case unfolding S all-decomposition-implies-def
proof
  fix x P level
  assume x: x ∈ set (get-all-marked-decomposition
    (fst (Propagated (- lit-of L) P # M, clauses S)))
  let ?M' = Propagated (- lit-of L) P # M

```

```

let ?hd = hd (get-all-marked-decomposition ?M')
let ?tl = tl (get-all-marked-decomposition ?M')
have x = ?hd  $\vee$  x  $\in$  set ?tl
  using x
  by (cases get-all-marked-decomposition ?M')
    auto
moreover {
  assume x': x  $\in$  set ?tl
  have L': Marked (lit-of L) () = L using marked by (case-tac L, auto)
  have x  $\in$  set (get-all-marked-decomposition (M' @ L # M))
    using x' get-all-marked-decomposition-except-last-choice-equal[of M' lit-of L P M]
    L' by (metis (no-types) M' list.set-sel(2) tl-Nil)
  then have case x of (Ls, seen)  $\Rightarrow$  ( $\lambda a.$  {#lit-of a#}) ' set Ls  $\cup$  set-mset (clauses S)
     $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set seen
    using marked IH by (case-tac L) (auto simp add: S all-decomposition-implies-def)
}
moreover {
  assume x': x = ?hd
  have tl: tl (get-all-marked-decomposition (M' @ L # M))  $\neq$  []
  proof -
    have f1:  $\bigwedge ms.$  length (get-all-marked-decomposition (M' @ ms))
      = length (get-all-marked-decomposition ms)
    by (simp add: M' get-all-marked-decomposition-remove-unmarked-length)
    have Suc (length (get-all-marked-decomposition M))  $\neq$  Suc 0
    by blast
    then show ?thesis
      using f1 marked by (metis (no-types) get-all-marked-decomposition.simps(1) length-tl
        list.sel(3) list.size(3) marked-lit.collapse(1))
  qed
  obtain M0' M0 where
    L0: hd (tl (get-all-marked-decomposition (M' @ L # M))) = (M0, M0')
    by (cases hd (tl (get-all-marked-decomposition (M' @ L # M))))
  have x'': x = (M0, Propagated ( $-$ lit-of L) P # M0')
    unfolding x' using get-all-marked-decomposition-last-choice tl M' L0
    by (metis marked marked-lit.collapse(1))
  obtain l-get-all-marked-decomposition where
    get-all-marked-decomposition (trail S) = (L # M, M') # (M0, M0') #
    l-get-all-marked-decomposition
    using get-all-marked-decomposition-backtrack-split extracted by (metis (no-types) L0 S
      hd-Cons-tl n tl)
  then have M = M0' @ M0 using get-all-marked-decomposition-hd-hd by fastforce
  then have IL': ( $\lambda a.$  {#lit-of a#}) ' set M0  $\cup$  set-mset (clauses S)
     $\cup$  ( $\lambda a.$  {#lit-of a#}) ' set M0'  $\models_{ps}$  {# $-$  lit-of L#}
    using IL by (simp add: Un-commute Un-left-commute image-Un)
  moreover have H: ( $\lambda a.$  {#lit-of a#}) ' set M0  $\cup$  set-mset (clauses S)
     $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set M0'
    using IH x'' unfolding all-decomposition-implies-def by (metis (no-types, lifting) L0 S
      list.set-sel(1) list.set-sel(2) old.prod.case tl tl-Nil)
  ultimately have case x of (Ls, seen)  $\Rightarrow$  ( $\lambda a.$  {#lit-of a#}) ' set Ls  $\cup$  set-mset (clauses S)
     $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set seen
    using true-clss-clss-left-right unfolding x'' by auto
}
ultimately show case x of (Ls, seen)  $\Rightarrow$ 
  ( $\lambda a.$  {#lit-of a#}) ' set Ls  $\cup$  set-mset (snd (?M', clauses S))
   $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set seen

```

unfolding *snd-conv* by *blast*  
 qed  
 qed

Lemma theorem 2.8.3 page 72 of CW

**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-marked-decomposition (trail S))*  
 and *atm-of ' lits-of (trail S)  $\subseteq$  atms-of-mu (clauses S)*  
 shows *set-mset (clauses S')  $\cup \{\{\#lit-of L\# \mid L. is-marked L \wedge L \in set (trail S')\}\}$*   
 $\models_{ps} (\lambda a. \{\#lit-of a\# \}) ' \bigcup (set ' snd ' set (get-all-marked-decomposition (trail S')))$   
 using *all-decomposition-implies-trail-is-implied*[*OF dpll<sub>W</sub>-propagate-is-conclusion*[*OF assms*]] .

Lemma theorem 2.8.4 page 72 of CW

**lemma** *only-propagated-vars-unsat*:  
 assumes *marked:  $\forall x \in set M. \neg is-marked x$*   
 and *DN:  $D \in N$  and  $D: M \models_{as} CNot D$*   
 and *inv: all-decomposition-implies N (get-all-marked-decomposition M)*  
 and *atm-incl: atm-of ' lits-of M  $\subseteq$  atms-of-m N*  
 shows *unsatisfiable N*  
**proof** (*rule ccontr*)  
 assume  $\neg unsatisfiable N$   
 then obtain *I* where  
 $I: I \models_s N$  and  
*cons: consistent-interp I* and  
*tot: total-over-m I N*  
 unfolding *satisfiable-def* by *auto*  
 then have *I-D:  $I \models D$*   
 using *DN* unfolding *true-clss-def* by *auto*  
  
 have *l0:  $\{\{\#lit-of L\# \mid L. is-marked L \wedge L \in set M\} = \{\}\}$*  using *marked* by *auto*  
 have *atms-of-m (N  $\cup (\lambda a. \{\#lit-of a\# \}) ' set M) = atms-of-m N$*   
 using *atm-incl* unfolding *atms-of-m-def lits-of-def* by *auto*  
  
 then have *total-over-m I (N  $\cup (\lambda a. \{\#lit-of a\# \}) ' (set M)$ )*  
 using *tot* unfolding *total-over-m-def* by *auto*  
 then have  $I \models_s (\lambda a. \{\#lit-of a\# \}) ' (set M)$   
 using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons I*  
 unfolding *true-clss-clss-def l0* by *auto*  
 then have *IM:  $I \models_s (\lambda a. \{\#lit-of a\# \}) ' set M$*  by *auto*  
 {  
 fix *K*  
 assume  $K \in \# D$   
 then have  $-K \in lits-of M$   
 by (*auto split: split-if-asm*)  
 intro: *allE*[*OF D*[*unfolded true-annots-def Ball-def*], of  $\{\#-K\# \}$ ]]  
 then have  $-K \in I$  using *IM true-clss-singleton-lit-of-implies-incl* by *fastforce*  
 }  
 then have  $\neg I \models D$  using *cons* unfolding *true-clss-def consistent-interp-def* by *auto*  
 then show *False* using *I-D* by *blast*  
 qed

**lemma** *dpll<sub>W</sub>-same-clauses*:  
 assumes *dpll<sub>W</sub> S S'*  
 shows *clauses S = clauses S'*



```

using assms by (induct rule: dpllW.induct, auto)

lemma rtrancpl-dpllW-inv:
  assumes rtrancpl dpllW S S'
  and inv: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
  and atm-incl: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-mu (clauses S)
  and consistent-interp (lits-of (trail S))
  and no-dup (trail S)
  shows all-decomposition-implies-m (clauses S') (get-all-marked-decomposition (trail S'))
  and atm-of ' lits-of (trail S')  $\subseteq$  atms-of-mu (clauses S')
  and clauses S = clauses S'
  and consistent-interp (lits-of (trail S'))
  and no-dup (trail S')
  using assms
proof (induct rule: rtrancpl-induct)
  case base
  show
    all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S)) and
    atm-of ' lits-of (trail S)  $\subseteq$  atms-of-mu (clauses S) and
    clauses S = clauses S and
    consistent-interp (lits-of (trail S)) and
    no-dup (trail S) using assms by auto
  next
  case (step S' S'') note dpllWStar = this(1) and IH = this(3,4,5,6,7) and
    dpllW = this(2)
  moreover
    assume
      inv: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S)) and
      atm-incl: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-mu (clauses S) and
      cons: consistent-interp (lits-of (trail S)) and
      no-dup (trail S)
    ultimately have decomp: all-decomposition-implies-m (clauses S')
      (get-all-marked-decomposition (trail S')) and
      atm-incl': atm-of ' lits-of (trail S')  $\subseteq$  atms-of-mu (clauses S') and
      snd: clauses S = clauses S' and
      cons': consistent-interp (lits-of (trail S')) and
      no-dup': no-dup (trail S') by blast+
    show clauses S = clauses S'' using dpllW-same-clauses[OF dpllW] snd by metis

  show all-decomposition-implies-m (clauses S'') (get-all-marked-decomposition (trail S''))
    using dpllW-propagate-is-conclusion[OF dpllW] decomp atm-incl' by auto
  show atm-of ' lits-of (trail S'')  $\subseteq$  atms-of-mu (clauses S'')
    using dpllW-vars-in-snd-inv[OF dpllW] atm-incl atm-incl' by auto
  show no-dup (trail S'') using dpllW-distinct-inv[OF dpllW] no-dup' dpllW by auto
  show consistent-interp (lits-of (trail S''))
    using cons' no-dup' dpllW-consistent-interp-inv[OF dpllW] by auto
qed

definition dpllW-all-inv S  $\equiv$ 
  (all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S)))
   $\wedge$  atm-of ' lits-of (trail S)  $\subseteq$  atms-of-mu (clauses S)
   $\wedge$  consistent-interp (lits-of (trail S))
   $\wedge$  no-dup (trail S)

lemma dpllW-all-inv-dest[dest]:

```

**assumes**  $dpll_W\text{-all-inv } S$   
**shows**  $\text{all-decomposition-implies-}m \text{ (clauses } S) \text{ (get-all-marked-decomposition (trail } S))}$   
**and**  $\text{atm-of ' lits-of (trail } S) \subseteq \text{atms-of-mu (clauses } S)}$   
**and**  $\text{consistent-interp (lits-of (trail } S)) \wedge \text{no-dup (trail } S)}$   
**using**  $\text{assms unfolding } dpll_W\text{-all-inv-def lits-of-def}$  **by**  $\text{auto}$

**lemma**  $\text{rtranclp-dpll}_W\text{-all-inv}$ :  
**assumes**  $\text{rtranclp } dpll_W \ S \ S'$   
**and**  $dpll_W\text{-all-inv } S$   
**shows**  $dpll_W\text{-all-inv } S'$   
**using**  $\text{assms rtranclp-dpll}_W\text{-inv[OF assms(1)] unfolding } dpll_W\text{-all-inv-def lits-of-def}$  **by**  $\text{blast}$

**lemma**  $dpll_W\text{-all-inv}$ :  
**assumes**  $dpll_W \ S \ S'$   
**and**  $dpll_W\text{-all-inv } S$   
**shows**  $dpll_W\text{-all-inv } S'$   
**using**  $\text{assms rtranclp-dpll}_W\text{-all-inv}$  **by**  $\text{blast}$

**lemma**  $\text{rtranclp-dpll}_W\text{-inv-starting-from-0}$ :  
**assumes**  $\text{rtranclp } dpll_W \ S \ S'$   
**and**  $\text{inv: trail } S = []$   
**shows**  $dpll_W\text{-all-inv } S'$   
**proof** –  
**have**  $dpll_W\text{-all-inv } S$   
**using**  $\text{assms unfolding all-decomposition-implies-def } dpll_W\text{-all-inv-def}$  **by**  $\text{auto}$   
**then show**  $?thesis$  **using**  $\text{rtranclp-dpll}_W\text{-all-inv[OF assms(1)]}$  **by**  $\text{blast}$   
**qed**

**lemma**  $dpll_W\text{-can-do-step}$ :  
**assumes**  $\text{consistent-interp (set } M)$   
**and**  $\text{distinct } M$   
**and**  $\text{atm-of ' (set } M) \subseteq \text{atms-of-mu } N$   
**shows**  $\text{rtranclp } dpll_W \ ([], N) \text{ (map } (\lambda M. \text{Marked } M \ ()) \ M, N)$   
**using**  $\text{assms}$   
**proof** ( $\text{induct } M$ )  
**case**  $\text{Nil}$   
**then show**  $?case$  **by**  $\text{auto}$   
**next**  
**case** ( $\text{Cons } L \ M$ )  
**then have**  $\text{undefined-lit (map } (\lambda M. \text{Marked } M \ ()) \ M) \ L$   
**unfolding**  $\text{defined-lit-def consistent-interp-def}$  **by**  $\text{auto}$   
**moreover have**  $\text{atm-of } L \in \text{atms-of-mu } N$  **using**  $\text{Cons.premis(3)}$  **by**  $\text{auto}$   
**ultimately have**  $dpll_W \text{ (map } (\lambda M. \text{Marked } M \ ()) \ M, N) \text{ (map } (\lambda M. \text{Marked } M \ ()) \ (L \# M), N)$   
**using**  $dpll_W.\text{decided}$  **by**  $\text{auto}$   
**moreover have**  $\text{consistent-interp (set } M)$  **and**  $\text{distinct } M$  **and**  $\text{atm-of ' set } M \subseteq \text{atms-of-mu } N$   
**using**  $\text{Cons.premis unfolding consistent-interp-def}$  **by**  $\text{auto}$   
**ultimately show**  $?case$  **using**  $\text{Cons.hyps}$  **by**  $\text{auto}$   
**qed**

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

**lemma**  $dpll_W\text{-strong-completeness}$ :

```

assumes set  $M \models_{sm} N$ 
and consistent-interp (set  $M$ )
and distinct  $M$ 
and atm-of ' (set  $M$ )  $\subseteq$  atms-of-mu  $N$ 
shows  $dpll_W^{**} ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$ 
and conclusive-dpllW-state (map ( $\lambda M. \text{Marked } M ()$ )  $M, N$ )
proof –
  show rtrancpl  $dpll_W ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$  using dpllW-can-do-step assms by auto
  have map ( $\lambda M. \text{Marked } M ()$ )  $M \models_{asm} N$  using assms(1) true-annots-marked-true-cls by auto
  then show conclusive-dpllW-state (map ( $\lambda M. \text{Marked } M ()$ )  $M, N$ )
    unfolding conclusive-dpllW-state-def by auto
qed

```

**lemma** *dpll<sub>W</sub>-sound*:

```

assumes
  rtrancpl  $dpll_W ([], N) (M, N)$  and
   $\forall S. \neg dpll_W (M, N) S$ 
shows  $M \models_{asm} N \longleftrightarrow \text{satisfiable } (set-mset\ N)$  (is  $?A \longleftrightarrow ?B$ )
proof
  let  $?M' = \text{lits-of } M$ 
  assume  $?A$ 
  then have  $?M' \models_{sm} N$  by (simp add: true-annots-true-cls)
  moreover have consistent-interp  $?M'$ 
    using rtrancpl-dpllW-inv-starting-from-0[OF assms(1)] by auto
  ultimately show  $?B$  by auto
next
  assume  $?B$ 
  show  $?A$ 
  proof (rule ccontr)
    assume  $n: \neg ?A$ 
    have  $(\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of-mu } N) \vee (\exists D \in \#N. M \models_{as} CNot\ D)$ 
    proof –
      obtain  $D :: 'a\ \text{clause}$  where  $D: D \in \# N$  and  $\neg M \models_a D$ 
      using  $n$  unfolding true-annots-def Ball-def by auto
      then have  $(\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of } D) \vee M \models_{as} CNot\ D$ 
      unfolding true-annots-def Ball-def CNot-def true-annot-def
      using atm-of-lit-in-atms-of true-annot-iff-marked-or-true-lit true-cls-def by blast
      then show  $?thesis$ 

      using  $D$  apply auto by (meson atms-of-atms-of-m-mono mem-set-mset-iff subset-eq)
    qed
  moreover {
    assume  $\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of-mu } N$ 
    then have False using assms(2) decided by fastforce
  }
  moreover {
    assume  $\exists D \in \#N. M \models_{as} CNot\ D$ 
    then obtain  $D$  where  $DN: D \in \# N$  and  $MD: M \models_{as} CNot\ D$  by auto
    {
      assume  $\forall l \in set\ M. \neg \text{is-marked } l$ 
      moreover have dpllW-all-inv ( $[], N$ )
      using assms unfolding all-decomposition-implies-def dpllW-all-inv-def by auto
      ultimately have unsatisfiable (set-mset  $N$ )
      using only-propagated-vars-unsat[of  $M\ D\ set-mset\ N$ ]  $DN\ MD$ 
    }
  }

```

```

      rtrancp-dpllW-all-inv[OF assms(1)] by force
    then have False using ‹?B› by blast
  }
  moreover {
    assume l: ∃ l ∈ set M. is-marked l
    then have False
      using backtrack[of (M, N) - - - D ] DN MD assms(2)
        backtrack-split-some-is-marked-then-snd-has-hd[OF l]
      by (metis backtrack-split-snd-hd-marked fst-conv list.distinct(1) list.sel(1) snd-conv)
  }
  ultimately have False by blast
}
ultimately show False by blast
qed
qed

```

### 16.3 Termination

**definition** *dpll<sub>W</sub>-mes*  $M\ n =$   
 $\text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } (1::\text{nat})) (\text{rev } M) @ \text{replicate } (n - \text{length } M) \ 3$

**lemma** *length-dpll<sub>W</sub>-mes*:  
**assumes**  $\text{length } M \leq n$   
**shows**  $\text{length } (\text{dpll}_W\text{-mes } M\ n) = n$   
**using** *assms* **unfolding** *dpll<sub>W</sub>-mes-def* **by** *auto*

**lemma** *distinctcard-atm-of-lits-of-eq-length*:  
**assumes** *no-dup S*  
**shows**  $\text{card } (\text{atm-of } \text{'lits-of } S) = \text{length } S$   
**using** *assms* **by** (*induct S*) (*auto simp add: image-image lits-of-def*)

**lemma** *dpll<sub>W</sub>-card-decrease*:  
**assumes** *dpll*:  $\text{dpll}_W\ S\ S'$  **and**  $\text{length } (\text{trail } S') \leq \text{card vars}$   
**and**  $\text{length } (\text{trail } S) \leq \text{card vars}$   
**shows**  $(\text{dpll}_W\text{-mes } (\text{trail } S') (\text{card vars}), \text{dpll}_W\text{-mes } (\text{trail } S) (\text{card vars}))$   
 $\in \text{lexn } \{(a, b). a < b\} (\text{card vars})$   
**using** *assms*

**proof** (*induct rule: dpll<sub>W</sub>.induct*)  
**case** (*propagate C L S*)  
**have**  $m: \text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$   
 $@ \text{replicate } (\text{card vars} - \text{length } (\text{trail } S)) \ 3$   
 $= \text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S)) @ 3$   
 $\# \text{replicate } (\text{card vars} - \text{Suc } (\text{length } (\text{trail } S))) \ 3$   
**using** *propagate.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*  
**then show** ?*case*  
**using** *propagate.prem[s(1)]* **unfolding** *dpll<sub>W</sub>-mes-def* **by** (*fastforce simp add: lexn-conv assms(2)*)

**next**

**case** (*decided S L*)  
**have**  $m: \text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$   
 $@ \text{replicate } (\text{card vars} - \text{length } (\text{trail } S)) \ 3$   
 $= \text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S)) @ 3$   
 $\# \text{replicate } (\text{card vars} - \text{Suc } (\text{length } (\text{trail } S))) \ 3$   
**using** *decided.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*  
**then show** ?*case*  
**using** *decided.prem* **unfolding** *dpll<sub>W</sub>-mes-def* **by** (*force simp add: lexn-conv assms(2)*)

**next**

```

case (backtrack S M' L M D)
have L: is-marked L using backtrack.hyps(2) by auto
have S: trail S = M' @ L # M
  using backtrack.hyps(1) backtrack-split-list-eq[of trail S] by auto
show ?case
  using backtrack.premis L unfolding dpllW-mes-def S by (fastforce simp add: lexn-conv assms(2))
qed

```

Proposition theorem 2.8.7 page 73 of CW

**lemma** *dpll<sub>W</sub>-card-decrease'*:

```

assumes dpll: dpllW S S'
and atm-incl: atm-of ' lits-of (trail S) ⊆ atms-of-mu (clauses S)
and no-dup: no-dup (trail S)
shows (dpllW-mes (trail S') (card (atms-of-mu (clauses S'))),
      dpllW-mes (trail S) (card (atms-of-mu (clauses S)))) ∈ lex {(a, b). a < b}

```

**proof** –

```

have finite (atms-of-mu (clauses S)) unfolding atms-of-m-def by auto
then have 1: length (trail S) ≤ card (atms-of-mu (clauses S))
  using distinctcard-atm-of-lit-of-eq-length[OF no-dup] atm-incl card-mono by metis

```

**moreover**

```

have no-dup': no-dup (trail S') using dpll dpllW-distinct-inv no-dup by blast
have SS': clauses S' = clauses S using dpll by (auto dest!: dpllW-same-clauses)
have atm-incl': atm-of ' lits-of (trail S') ⊆ atms-of-mu (clauses S')
  using atm-incl dpll dpllW-vars-in-snd-inv[OF dpll] by force
have finite (atms-of-mu (clauses S'))
  unfolding atms-of-m-def by auto
then have 2: length (trail S') ≤ card (atms-of-mu (clauses S'))
  using distinctcard-atm-of-lit-of-eq-length[OF no-dup'] atm-incl' card-mono SS' by metis

```

```

ultimately have (dpllW-mes (trail S') (card (atms-of-mu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-mu (clauses S'))))
  ∈ lexn {(a, b). a < b} (card (atms-of-mu (clauses S)))
  using dpllW-card-decrease[OF assms(1), of atms-of-mu (clauses S)] by blast
then have (dpllW-mes (trail S') (card (atms-of-mu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-mu (clauses S')))) ∈ lex {(a, b). a < b}
  unfolding lex-def by auto
then show (dpllW-mes (trail S') (card (atms-of-mu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-mu (clauses S')))) ∈ lex {(a, b). a < b}
  using dpllW-same-clauses[OF assms(1)] by auto
qed

```

**lemma** *wf-lexn*: wf (lexn {(a, b). (a::nat) < b} (card (atms-of-mu (clauses S))))

**proof** –

```

have m: {(a, b). a < b} = measure id by auto
show ?thesis apply (rule wf-lexn) unfolding m by auto
qed

```

**lemma** *dpll<sub>W</sub>-wf*:

```

wf {(S', S). dpllW-all-inv S ∧ dpllW S S'}
apply (rule wf-wf-if-measure'[OF wf-lex-less, of - -
  λS. dpllW-mes (trail S) (card (atms-of-mu (clauses S)))])
using dpllW-card-decrease' by fast

```

**lemma** *dpll<sub>W</sub>-trancpl-star-commute*:  
 $\{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+ = \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{trancpl } \text{dpll}_W S S'\}$   
 (is ?A = ?B)

**proof**

```

{ fix S S'
  assume (S, S') ∈ ?A
  then have (S, S') ∈ ?B
    by (induct rule: trancpl.induct, auto)
}
then show ?A ⊆ ?B by blast
{ fix S S'
  assume (S, S') ∈ ?B
  then have dpllW++ S' S and dpllW-all-inv S' by auto
  then have (S, S') ∈ ?A
    proof (induct rule: trancpl.induct)
      case r-into-trancpl
      then show ?case by (simp-all add: r-into-trancpl')
    next
      case (trancpl-into-trancpl S S' S'')
      then have (S', S) ∈ {a. case a of (S', S) ⇒ dpllW-all-inv S ∧ dpllW S S'}+ by blast
      moreover have dpllW-all-inv S'
        using rtrancpl-dpllW-all-inv[OF trancpl-into-rtrancpl[OF trancpl-into-trancpl.hyps(1)]]
        trancpl-into-trancpl.prem by auto
      ultimately have (S'', S') ∈ {(pa, p). dpllW-all-inv p ∧ dpllW p pa}+
        using ⟨dpllW-all-inv S'⟩ trancpl-into-trancpl.hyps(3) by blast
      then show ?case
        using ⟨(S', S) ∈ {a. case a of (S', S) ⇒ dpllW-all-inv S ∧ dpllW S S'}+⟩ by auto
    qed
  }
then show ?B ⊆ ?A by blast
qed

```

**lemma** *dpll<sub>W</sub>-wf-trancpl*: wf {(S', S). dpll<sub>W</sub>-all-inv S ∧ dpll<sub>W</sub><sup>++</sup> S S'}  
 unfolding dpll<sub>W</sub>-trancpl-star-commute[symmetric] by (simp add: dpll<sub>W</sub>-wf wf-trancpl)

**lemma** *dpll<sub>W</sub>-wf-plus*:  
 shows wf {(S', (□, N)) | S'. dpll<sub>W</sub><sup>++</sup> (□, N) S'} (is wf ?P)  
 apply (rule wf-subset[OF dpll<sub>W</sub>-wf-trancpl, of ?P])  
 using assms unfolding dpll<sub>W</sub>-all-inv-def by auto

## 16.4 Final States

**lemma** *dpll<sub>W</sub>-no-more-step-is-a-conclusive-state*:

assumes  $\forall S'. \neg \text{dpll}_W S S'$   
 shows conclusive-dpll<sub>W</sub>-state S

**proof** —

```

have vars: ∀ s ∈ atms-of-mu (clauses S). s ∈ atm-of ' lits-of (trail S)
proof (rule ccontr)
  assume ¬ (∀ s ∈ atms-of-mu (clauses S). s ∈ atm-of ' lits-of (trail S))
  then obtain L where
    L-in-atms: L ∈ atms-of-mu (clauses S) and
    L-notin-trail: L ∉ atm-of ' lits-of (trail S) by metis
  obtain L' where L': atm-of L' = L by (meson literal.sel(2))
  then have undefined-lit (trail S) L'
    unfolding Marked-Propagated-in-iff-in-lits-of by (metis L-notin-trail atm-of-uminus imageI)
  then show False using dpllW.decided assms(1) L-in-atms L' by blast

```

```

qed
show ?thesis
proof (rule ccontr)
  assume not-final:  $\neg$  ?thesis
  then have
     $\neg$  trail S  $\models_{asm}$  clauses S and
     $(\exists L \in \text{set } (\text{trail } S). \text{is-marked } L) \vee (\forall C \in \# \text{clauses } S. \neg \text{trail } S \models_{as} CNot\ C)$ 
    unfolding conclusive-dpllW-state-def by auto
  moreover {
    assume  $\exists L \in \text{set } (\text{trail } S). \text{is-marked } L$ 
    then obtain L M' M where L: backtrack-split (trail S) = (M', L # M)
      using backtrack-split-some-is-marked-then-snd-has-hd by blast
    obtain D where D  $\in \#$  clauses S and  $\neg$  trail S  $\models_a$  D
      using  $\langle \neg \text{trail } S \models_{asm} \text{clauses } S \rangle$  unfolding true-annots-def by auto
    then have  $\forall s \in \text{atms-of-m } \{D\}. s \in \text{atm-of ' lits-of } (\text{trail } S)$ 
      using vars unfolding atms-of-m-def by auto
    then have trail S  $\models_{as}$  CNot D
      using all-variables-defined-not-imply-cnot[of D]  $\langle \neg \text{trail } S \models_a D \rangle$  by auto
    moreover have is-marked L
      using L by (metis backtrack-split-snd-hd-marked list.distinct(1) list.sel(1) snd-conv)
    ultimately have False
      using assms(1) dpllW.backtrack L  $\langle D \in \# \text{clauses } S \rangle \langle \text{trail } S \models_{as} CNot\ D \rangle$  by blast
  }
  moreover {
    assume tr:  $\forall C \in \# \text{clauses } S. \neg \text{trail } S \models_{as} CNot\ C$ 
    obtain C where C-in-cl: C  $\in \#$  clauses S and trC:  $\neg \text{trail } S \models_a C$ 
      using  $\langle \neg \text{trail } S \models_{asm} \text{clauses } S \rangle$  unfolding true-annots-def by auto
    have  $\forall s \in \text{atms-of-m } \{C\}. s \in \text{atm-of ' lits-of } (\text{trail } S)$ 
      using vars  $\langle C \in \# \text{clauses } S \rangle$  unfolding atms-of-m-def by auto
    then have trail S  $\models_{as}$  CNot C
      by (meson C-in-cl tr trC all-variables-defined-not-imply-cnot)
    then have False using tr C-in-cl by auto
  }
  ultimately show False by blast
qed
qed

lemma dpllW-conclusive-state-correct:
  assumes dpllW** ([], N) (M, N) and conclusive-dpllW-state (M, N)
  shows M  $\models_{asm}$  N  $\longleftrightarrow$  satisfiable (set-mset N) (is ?A  $\longleftrightarrow$  ?B)
proof
  let ?M' = lits-of M
  assume ?A
  then have ?M'  $\models_{sm}$  N by (simp add: true-annots-true-cl)
  moreover have consistent-interp ?M'
    using rtranclp-dpllW-inv-starting-from-0[OF assms(1)] by auto
  ultimately show ?B by auto
next
  assume ?B
  show ?A
  proof (rule ccontr)
    assume n:  $\neg$  ?A
    have no-mark:  $\forall L \in \text{set } M. \neg \text{is-marked } L \ \exists C \in \# N. M \models_{as} CNot\ C$ 
      using n assms(2) unfolding conclusive-dpllW-state-def by auto
    moreover obtain D where DN: D  $\in \#$  N and MD: M  $\models_{as}$  CNot D using no-mark by auto

```

```

ultimately have unsatisfiable (set-mset N)
  using only-propagated-vars-unsat rtrancpl-dpllW-all-inv[OF assms(1)]
  unfolding dpllW-all-inv-def by force
  then show False using ⟨?B⟩ by blast
qed
qed

```

## 16.5 Link with NOT's DPLL

interpretation  $dpll_{W-NOT}$ : *dpll-with-backtrack* .

```

lemma state-eqNOT-iff-eq[iff, simp]: dpllW-NOT.state-eqNOT S T  $\longleftrightarrow$  S = T
  unfolding dpllW-NOT.state-eqNOT-def by (cases S, cases T) auto

```

```

declare dpllW-NOT.state-simpNOT[simp del]

```

```

lemma dpllW-dpllW-bj:
  assumes inv: dpllW-all-inv S and dpll: dpllW S T
  shows dpllW-NOT.dpll-bj S T
  using dpll inv
  apply (induction rule: dpllW.induct)
    using dpllW-NOT.dpll-bj.simps apply fastforce
    using dpllW-NOT.bj-decideNOT apply fastforce
  apply (frule dpllW-NOT.backtrack.intros[of - - - -], simp-all)
  apply (rule dpllW-NOT.dpll-bj.bj-backjump)
  apply (rule dpllW-NOT.backtrack-is-backjump'',
    simp-all add: dpllW-all-inv-def)
done

```

```

lemma dpllW-bj-dpll:
  assumes inv: dpllW-all-inv S and dpll: dpllW-NOT.dpll-bj S T
  shows dpllW S T
  using dpll
  apply (induction rule: dpllW-NOT.dpll-bj.induct)
    apply (elim dpllW-NOT.decideE, cases S)
    using decided apply fastforce
    apply (elim dpllW-NOT.propagateE, cases S)
    using dpllW.simps apply fastforce
    apply (elim dpllW-NOT.backjumpE, cases S)
  by (simp add: dpllW.simps dpll-with-backtrack.backtrack.simps)

```

```

lemma rtrancpl-dpllW-rtrancpl-dpllW-NOT:
  assumes dpllW** S T and dpllW-all-inv S
  shows dpllW-NOT.dpll-bj** S T
  using assms apply (induction)
  apply simp
  by (auto intro: rtrancpl-dpllW-all-inv dpllW-dpllW-bj rtrancpl.rtrancpl-into-rtrancpl)

```

```

lemma rtrancpl-dpll-rtrancpl-dpllW:
  assumes dpllW-NOT.dpll-bj** S T and dpllW-all-inv S
  shows dpllW** S T
  using assms apply (induction)
  apply simp
  by (auto intro: dpllW-bj-dpll rtrancpl.rtrancpl-into-rtrancpl rtrancpl-dpllW-all-inv)

```

```

lemma dpll-conclusive-state-correctness:

```



```

assumes  $dpll_W\text{-}NOT.dpll\text{-}bj^{**} ([], N) (M, N)$  and  $conclusive\text{-}dpll_W\text{-}state (M, N)$ 
shows  $M \models_{asm} N \longleftrightarrow satisfiable (set\text{-}mset N)$ 
proof –
  have  $dpll_W\text{-}all\text{-}inv ([], N)$ 
    unfolding  $dpll_W\text{-}all\text{-}inv\text{-}def$  by auto
  show ?thesis
    apply (rule  $dpll_W\text{-}conclusive\text{-}state\text{-}correct$ )
      apply (simp add:  $\langle dpll_W\text{-}all\text{-}inv ([], N) \rangle assms(1) rtranclp\text{-}dpll\text{-}rtranclp\text{-}dpll_W$ )
      using  $assms(2)$  by simp
qed

end
theory CDCL-W-Level
imports Partial-Annotated-Clausal-Logic
begin

```

### 16.5.1 Level of literals and clauses

Getting the level of a variable, implies that the list has to be reversed. Here is the funtion after reversing.

```

fun  $get\text{-}rev\text{-}level :: 'v\ literal \Rightarrow nat \Rightarrow ('v, nat, 'a) marked\text{-}lits \Rightarrow nat$  where
 $get\text{-}rev\text{-}level - [] = 0$  |
 $get\text{-}rev\text{-}level L\ n\ (Marked\ l\ level\ \# Ls) =$ 
   $(if\ atm\text{-}of\ l = atm\text{-}of\ L\ then\ level\ else\ get\text{-}rev\text{-}level\ L\ level\ Ls) |$ 
 $get\text{-}rev\text{-}level L\ n\ (Propagated\ l\ -\ \# Ls) =$ 
   $(if\ atm\text{-}of\ l = atm\text{-}of\ L\ then\ n\ else\ get\text{-}rev\text{-}level\ L\ n\ Ls)$ 

```

**abbreviation**  $get\text{-}level\ L\ M \equiv get\text{-}rev\text{-}level\ L\ 0\ (rev\ M)$

**lemma**  $get\text{-}rev\text{-}level\text{-}uminus[simp]$ :  $get\text{-}rev\text{-}level\ (-L)\ n\ M = get\text{-}rev\text{-}level\ L\ n\ M$   
**by** (*induct*  $M$  *arbitrary*:  $n$  *rule*:  $get\text{-}rev\text{-}level.induct$ ) *auto*

**lemma**  $atm\text{-}of\text{-}notin\text{-}get\text{-}rev\text{-}level\text{-}eq\text{-}0[simp]$ :  
**assumes**  $atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ M$   
**shows**  $get\text{-}rev\text{-}level\ L\ n\ M = 0$   
**using**  $assms$  **apply** (*induct*  $M$  *arbitrary*:  $n$ , *simp*)  
**by** (*case-tac*  $a$ ) *auto*

**lemma**  $get\text{-}rev\text{-}level\text{-}ge\text{-}0\text{-}atm\text{-}of\text{-}in$ :  
**assumes**  $get\text{-}rev\text{-}level\ L\ n\ M > n$   
**shows**  $atm\text{-}of\ L \in atm\text{-}of\ ' lits\text{-}of\ M$   
**using**  $assms$  **apply** (*induct*  $M$  *arbitrary*:  $n$ , *simp*)  
**by** (*case-tac*  $a$ ) *fastforce*+

In  $get\text{-}rev\text{-}level$  (resp.  $get\text{-}level$ ), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

**lemma**  $get\text{-}rev\text{-}level\text{-}skip[simp]$ :  
**assumes**  $atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ M$   
**shows**  $get\text{-}rev\text{-}level\ L\ n\ (M @ Marked\ K\ i\ \# M') = get\text{-}rev\text{-}level\ L\ i\ (Marked\ K\ i\ \# M')$   
**using**  $assms$  **apply** (*induct*  $M$  *arbitrary*:  $n\ i$ , *simp*)  
**by** (*case-tac*  $a$ ) *auto*

**lemma**  $get\text{-}rev\text{-}level\text{-}notin\text{-}end[simp]$ :  
**assumes**  $atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ M'$   
**shows**  $get\text{-}rev\text{-}level\ L\ n\ (M @ M') = get\text{-}rev\text{-}level\ L\ n\ M$

**using** *assms* **apply** (*induct M arbitrary: n, simp*)  
**by** (*case-tac a*) *auto*

If the literal is at the beginning, then the end can be skipped

**lemma** *get-rev-level-skip-end*[*simp*]:  
**assumes** *atm-of L*  $\in$  *atm-of* ' *lits-of M*  
**shows** *get-rev-level L n* (*M* @ *M'*) = *get-rev-level L n M*  
**using** *assms* **apply** (*induct M arbitrary: n, simp*)  
**by** (*case-tac a*) *auto*

**lemma** *get-level-skip-beginning*:  
**assumes** *atm-of L'*  $\neq$  *atm-of* (*lit-of K*)  
**shows** *get-level L'* (*K* # *M*) = *get-level L' M*  
**using** *assms* **by** *auto*

**lemma** *get-level-skip-beginning-not-marked-rev*:  
**assumes** *atm-of L*  $\notin$  *atm-of* ' *lit-of* ' (*set S*)  
**and**  $\forall s \in \text{set } S. \neg \text{is-marked } s$   
**shows** *get-level L* (*M* @ *rev S*) = *get-level L M*  
**using** *assms* **by** (*induction S rule: marked-lit-list-induct*) *auto*

**lemma** *get-level-skip-beginning-not-marked*[*simp*]:  
**assumes** *atm-of L*  $\notin$  *atm-of* ' *lit-of* ' (*set S*)  
**and**  $\forall s \in \text{set } S. \neg \text{is-marked } s$   
**shows** *get-level L* (*M* @ *S*) = *get-level L M*  
**using** *get-level-skip-beginning-not-marked-rev*[*of L rev S M*] *assms* **by** *auto*

**lemma** *get-rev-level-skip-beginning-not-marked*[*simp*]:  
**assumes** *atm-of L*  $\notin$  *atm-of* ' *lit-of* ' (*set S*)  
**and**  $\forall s \in \text{set } S. \neg \text{is-marked } s$   
**shows** *get-rev-level L 0* (*rev S* @ *rev M*) = *get-level L M*  
**using** *get-level-skip-beginning-not-marked-rev*[*of L rev S M*] *assms* **by** *auto*

**lemma** *get-level-skip-in-all-not-marked*:  
**fixes** *M* :: ('a, nat, 'b) *marked-lit list* **and** *L* :: 'a *literal*  
**assumes**  $\forall m \in \text{set } M. \neg \text{is-marked } m$   
**and** *atm-of L*  $\in$  *atm-of* ' *lit-of* ' (*set M*)  
**shows** *get-rev-level L n M* = *n*

**proof** –  
**show** ?*thesis*  
**using** *assms* **by** (*induction M rule: marked-lit-list-induct*) *auto*  
**qed**

**lemma** *get-level-skip-all-not-marked*[*simp*]:  
**fixes** *M*  
**defines** *M'*  $\equiv$  *rev M*  
**assumes**  $\forall m \in \text{set } M. \neg \text{is-marked } m$   
**shows** *get-level L M* = 0

**proof** –  
**have** *M*: *M* = *rev M'*  
**unfolding** *M'-def* **by** *auto*  
**show** ?*thesis*  
**using** *assms* **unfolding** *M* **by** (*induction M' rule: marked-lit-list-induct*) *auto*  
**qed**

**abbreviation**  $MMax\ M \equiv Max\ (set-mset\ M)$

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

**definition**  $get-maximum-level :: 'a\ literal\ multiset \Rightarrow ('a, nat, 'b)\ marked-lit\ list \Rightarrow nat$   
**where**

$get-maximum-level\ D\ M = MMax\ (\{\#0\#\} + image-mset\ (\lambda L. get-level\ L\ M)\ D)$

**lemma**  $get-maximum-level-ge-get-level$ :

$L \in \# D \implies get-maximum-level\ D\ M \geq get-level\ L\ M$

**unfolding**  $get-maximum-level-def$  **by**  $auto$

**lemma**  $get-maximum-level-empty[simp]$ :

$get-maximum-level\ \{\#\}\ M = 0$

**unfolding**  $get-maximum-level-def$  **by**  $auto$

**lemma**  $get-maximum-level-exists-lit-of-max-level$ :

$D \neq \{\#\} \implies \exists L \in \# D. get-level\ L\ M = get-maximum-level\ D\ M$

**unfolding**  $get-maximum-level-def$

**apply**  $(induct\ D)$

**apply**  $simp$

**by**  $(case-tac\ D = \{\#\})\ (auto\ simp\ add: max-def)$

**lemma**  $get-maximum-level-empty-list[simp]$ :

$get-maximum-level\ D\ [] = 0$

**unfolding**  $get-maximum-level-def$  **by**  $(simp\ add: image-constant-conv)$

**lemma**  $get-maximum-level-single[simp]$ :

$get-maximum-level\ \{\#L\#\}\ M = get-level\ L\ M$

**unfolding**  $get-maximum-level-def$  **by**  $simp$

**lemma**  $get-maximum-level-plus$ :

$get-maximum-level\ (D + D')\ M = max\ (get-maximum-level\ D\ M)\ (get-maximum-level\ D'\ M)$

**by**  $(induct\ D)\ (auto\ simp\ add: get-maximum-level-def)$

**lemma**  $get-maximum-level-exists-lit$ :

**assumes**  $n: n > 0$

**and**  $max: get-maximum-level\ D\ M = n$

**shows**  $\exists L \in \# D. get-level\ L\ M = n$

**proof** –

**have**  $f: finite\ (insert\ 0\ ((\lambda L. get-level\ L\ M)\ ' set-mset\ D))$  **by**  $auto$

**hence**  $n \in ((\lambda L. get-level\ L\ M)\ ' set-mset\ D)$

**using**  $n\ max\ Max-in[OF\ f]$  **unfolding**  $get-maximum-level-def$  **by**  $simp$

**thus**  $\exists L \in \# D. get-level\ L\ M = n$  **by**  $auto$

**qed**

**lemma**  $get-maximum-level-skip-first[simp]$ :

**assumes**  $atm-of\ L \notin atms-of\ D$

**shows**  $get-maximum-level\ D\ (Propagated\ L\ C\ \# M) = get-maximum-level\ D\ M$

**using**  $assms$  **unfolding**  $get-maximum-level-def\ atms-of-def$

$atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set$

**by**  $(smt\ atm-of-in-atm-of-set-in-uminus\ get-level-skip-beginning\ image-iff\ marked-lit.sel(2)\ multiset.map-cong0)$

**lemma** *get-maximum-level-skip-beginning*:  
**assumes**  $DH$ :  $\text{atms-of } D \subseteq \text{atm-of 'lits-of } H$   
**shows**  $\text{get-maximum-level } D (c @ \text{Marked } Kh \ i \ \# \ H) = \text{get-maximum-level } D \ H$   
**proof** –  
**have**  $(\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } H @ \text{Marked } Kh \ i \ \# \ \text{rev } c)) \text{ 'set-mset } D$   
 $= (\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } H)) \text{ 'set-mset } D$   
**using**  $DH$  **unfolding** *atms-of-def*  
**by** (*metis* (*no-types*, *lifting*) *get-rev-level-skip-end image-cong image-subset-iff lits-of-rev*) +  
**thus** ?thesis **using**  $DH$  **unfolding** *get-maximum-level-def* **by** *auto*  
**qed**

**lemma** *get-maximum-level-D-single-propagated*:  
 $\text{get-maximum-level } D [\text{Propagated } x21 \ x22] = 0$   
**proof** –  
**have**  $A$ :  $\text{insert } 0 \ ((\lambda L. \ 0) \text{ 'set-mset } D \cap \{L. \text{atm-of } x21 = \text{atm-of } L\})$   
 $\cup (\lambda L. \ 0) \text{ 'set-mset } D \cap \{L. \text{atm-of } x21 \neq \text{atm-of } L\}) = \{0\}$   
**by** *auto*  
**show** ?thesis **unfolding** *get-maximum-level-def* **by** (*simp add: A*)  
**qed**

**lemma** *get-maximum-level-skip-notin*:  
**assumes**  $D$ :  $\forall L \in \#D. \text{atm-of } L \in \text{atm-of 'lits-of } M$   
**shows**  $\text{get-maximum-level } D \ M = \text{get-maximum-level } D (\text{Propagated } x21 \ x22 \ \# \ M)$   
**proof** –  
**have**  $A$ :  $(\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } M @ [\text{Propagated } x21 \ x22])) \text{ 'set-mset } D$   
 $= (\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } M)) \text{ 'set-mset } D$   
**using**  $D$  **by** (*auto intro!*: *image-cong simp add: lits-of-def*)  
**show** ?thesis **unfolding** *get-maximum-level-def* **by** (*auto simp add: A*)  
**qed**

**lemma** *get-maximum-level-skip-un-marked-not-present*:  
**assumes**  $\forall L \in \#D. \text{atm-of } L \in \text{atm-of 'lits-of } aa$  **and**  
 $\forall m \in \text{set } M. \neg \text{is-marked } m$   
**shows**  $\text{get-maximum-level } D \ aa = \text{get-maximum-level } D (M @ aa)$   
**using** *assms* **apply** (*induction M*)  
**apply** *simp*  
**by** (*case-tac a*) (*auto intro!*: *get-maximum-level-skip-notin[of D - @ aa]* *simp add: image-Un*)

**fun** *get-maximum-possible-level*:: (*'b*, *nat*, *'c*) *marked-lit list*  $\Rightarrow$  *nat* **where**  
 $\text{get-maximum-possible-level } [] = 0 \mid$   
 $\text{get-maximum-possible-level } (\text{Marked } K \ i \ \# \ l) = \max i (\text{get-maximum-possible-level } l) \mid$   
 $\text{get-maximum-possible-level } (\text{Propagated } - \ \# \ l) = \text{get-maximum-possible-level } l$

**lemma** *get-maximum-possible-level-append[simp]*:  
 $\text{get-maximum-possible-level } (M @ M')$   
 $= \max (\text{get-maximum-possible-level } M) (\text{get-maximum-possible-level } M')$   
**apply** (*induct M, simp*) **by** (*case-tac a, auto*)

**lemma** *get-maximum-possible-level-rev[simp]*:  
 $\text{get-maximum-possible-level } (\text{rev } M) = \text{get-maximum-possible-level } M$   
**apply** (*induct M, simp*) **by** (*case-tac a, auto*)

**lemma** *get-maximum-possible-level-ge-get-rev-level*:  
 $\max (\text{get-maximum-possible-level } M) \ i \geq \text{get-rev-level } L \ i \ M$   
**apply** (*induct M arbitrary: i*)

```

  apply simp
by (case-tac a) (auto simp add: le-max-iff-disj)

lemma get-maximum-possible-level-ge-get-level[simp]:
  get-maximum-possible-level M ≥ get-level L M
  using get-maximum-possible-level-ge-get-rev-level[of - 0 rev -] by auto

lemma get-maximum-possible-level-ge-get-maximum-level[simp]:
  get-maximum-possible-level M ≥ get-maximum-level D M
  using get-maximum-level-exists-lit-of-max-level unfolding Bex-mset-def
  by (metis get-maximum-level-empty get-maximum-possible-level-ge-get-level le0)

fun get-all-mark-of-propagated where
  get-all-mark-of-propagated [] = [] |
  get-all-mark-of-propagated (Marked - - # 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
  apply (induct A, simp)
  by (case-tac a) auto



### 16.5.2 Properties about the levels



fun get-all-levels-of-marked :: ('b, 'a, 'c) marked-lit list ⇒ 'a list where
  get-all-levels-of-marked [] = [] |
  get-all-levels-of-marked (Marked l level # Ls) = level # get-all-levels-of-marked Ls |
  get-all-levels-of-marked (Propagated - - # Ls) = get-all-levels-of-marked Ls

lemma get-all-levels-of-marked-nil-iff-not-is-marked:
  get-all-levels-of-marked xs = [] ⟷ (∀ x ∈ set xs. ¬is-marked x)
  using assms by (induction xs rule: marked-lit-list-induct) auto

lemma get-all-levels-of-marked-cons:
  get-all-levels-of-marked (a # b) =
    (if is-marked a then [level-of a] else []) @ get-all-levels-of-marked b
  by (case-tac a) simp-all

lemma get-all-levels-of-marked-append[simp]:
  get-all-levels-of-marked (a @ b) = get-all-levels-of-marked a @ get-all-levels-of-marked b
  by (induct a) (simp-all add: get-all-levels-of-marked-cons)

lemma in-get-all-levels-of-marked-iff-decomp:
  i ∈ set (get-all-levels-of-marked M) ⟷ (∃ c K c'. M = c @ Marked K i # c') (is ?A ⟷ ?B)
proof
  assume ?B
  thus ?A by auto
next
  assume ?A
  thus ?B
  apply (induction M rule: marked-lit-list-induct)
  apply auto[]
  apply (metis append-Cons append-Nil get-all-levels-of-marked.simps(2) set-ConsD)
  by (metis append-Cons get-all-levels-of-marked.simps(3))
qed

```

**lemma** *get-rev-level-less-max-get-all-levels-of-marked*:  
*get-rev-level*  $L$   $n$   $M \leq \text{Max} (\text{set } (n \# \text{get-all-levels-of-marked } M))$   
**by** (*induct*  $M$  *arbitrary*:  $n$  *rule*: *get-all-levels-of-marked.induct*)  
*(simp-all add: max.coboundedI2)*

**lemma** *get-rev-level-ge-min-get-all-levels-of-marked*:  
**assumes** *atm-of*  $L \in \text{atm-of } \text{' } \text{ lits-of } M$   
**shows** *get-rev-level*  $L$   $n$   $M \geq \text{Min} (\text{set } (n \# \text{get-all-levels-of-marked } M))$   
**using** *assms* **by** (*induct*  $M$  *arbitrary*:  $n$  *rule*: *get-all-levels-of-marked.induct*)  
*(auto simp add: min-le-iff-disj)*

**lemma** *get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked[simp]*:  
*get-all-levels-of-marked* (*rev*  $M$ ) = *rev* (*get-all-levels-of-marked*  $M$ )  
**by** (*induct*  $M$  *rule*: *get-all-levels-of-marked.induct*)  
*(simp-all add: max.coboundedI2)*

**lemma** *get-maximum-possible-level-max-get-all-levels-of-marked*:  
*get-maximum-possible-level*  $M = \text{Max} (\text{insert } 0 (\text{set } (\text{get-all-levels-of-marked } M)))$   
**apply** (*induct*  $M$ , *simp*)  
**by** (*case-tac*  $a$ ) (*case-tac* *set* (*get-all-levels-of-marked*  $M$ ) =  $\{\}$ , *auto*)

**lemma** *get-rev-level-in-levels-of-marked*:  
*get-rev-level*  $L$   $n$   $M \in \{0, n\} \cup \text{set } (\text{get-all-levels-of-marked } M)$   
**apply** (*induction*  $M$  *arbitrary*:  $n$ )  
**apply** *auto*[1]  
**by** (*case-tac*  $a$ )  
*(force simp add: atm-of-eq-atm-of)+*

**lemma** *get-rev-level-in-atms-in-levels-of-marked*:  
*atm-of*  $L \in \text{atm-of } \text{' } (\text{ lits-of } M) \implies \text{get-rev-level } L$   $n$   $M \in \{n\} \cup \text{set } (\text{get-all-levels-of-marked } M)$   
**apply** (*induction*  $M$  *arbitrary*:  $n$ , *simp*)  
**by** (*case-tac*  $a$ )  
*(auto simp add: atm-of-eq-atm-of)*

**lemma** *get-all-levels-of-marked-no-marked*:  
 $(\forall l \in \text{set } Ls. \neg \text{is-marked } l) \iff \text{get-all-levels-of-marked } Ls = []$   
**by** (*induction*  $Ls$ ) (*auto simp add: get-all-levels-of-marked-cons*)

**lemma** *get-level-in-levels-of-marked*:  
*get-level*  $L$   $M \in \{0\} \cup \text{set } (\text{get-all-levels-of-marked } M)$   
**using** *get-rev-level-in-levels-of-marked*[*of*  $L$   $0$  *rev*  $M$ ] **by** *auto*

The zero is here to avoid empty-list issues with *last*:

**lemma** *get-level-get-rev-level-get-all-levels-of-marked*:  
**assumes** *atm-of*  $L \notin \text{atm-of } \text{' } (\text{ lits-of } M)$   
**shows** *get-level*  $L$  ( $K @ M$ ) = *get-rev-level*  $L$  (*last* ( $0 \# \text{get-all-levels-of-marked } (\text{rev } M)$ ))  
*(rev*  $K$ )  
**using** *assms*  
**proof** (*induct*  $M$  *arbitrary*:  $K$ )  
**case** *Nil*  
**thus** ?*case* **by** *auto*  
**next**  
**case** (*Cons*  $a$   $M$ )  
**hence**  $H$ :  $\bigwedge K. \text{get-level } L (K @ M)$

```

    = get-rev-level L (last (0 # get-all-levels-of-marked (rev M))) (rev K)
  by auto
have get-level L ((K @ [a])@ M)
  = get-rev-level L (last (0 # get-all-levels-of-marked (rev M))) (a # rev K)
  using H[of K @ [a]] by simp
thus ?case using Cons(2) by (case-tac a) auto
qed

```

**lemma** *get-rev-level-can-skip-correctly-ordered:*

```

  assumes no-dup M
  and atm-of L  $\notin$  atm-of ' (lits-of M)
  and get-all-levels-of-marked M = rev [Suc 0.. $\leq$ Suc (length (get-all-levels-of-marked M))]
  shows get-rev-level L 0 (rev M @ K) = get-rev-level L (length (get-all-levels-of-marked M)) K
  using assms
proof (induct M arbitrary: K)
  case Nil
  thus ?case by simp
next
  case (Cons a M K)
  show ?case
  proof (case-tac a)
    fix L' i
    assume a: a = Marked L' i
    have i: i = Suc (length (get-all-levels-of-marked M))
    and get-all-levels-of-marked M = rev [Suc 0.. $\leq$ Suc (length (get-all-levels-of-marked M))]
      using Cons.prem(3) unfolding a by auto
    hence get-rev-level L 0 (rev M @ (a # K))
      = get-rev-level L (length (get-all-levels-of-marked M)) (a # K)
      using Cons.hyps Cons.prem by auto
    thus ?case using Cons.prem(2) unfolding a i by auto
  next
    fix L' D
    assume a: a = Propagated L' D
    have get-all-levels-of-marked M = rev [Suc 0.. $\leq$ Suc (length (get-all-levels-of-marked M))]
      using Cons.prem(3) unfolding a by auto
    hence get-rev-level L 0 (rev M @ (a # K))
      = get-rev-level L (length (get-all-levels-of-marked M)) (a # K)
      using Cons by auto
    thus ?case using Cons.prem(2) unfolding a by auto
  qed
qed

```

**lemma** *get-level-skip-beginning-hd-get-all-levels-of-marked:*

```

  assumes atm-of L  $\notin$  atm-of ' lits-of S
  and get-all-levels-of-marked S  $\neq$  []
  shows get-level L (M @ S) = get-rev-level L (hd (get-all-levels-of-marked S)) (rev M)
  using assms
proof (induction S arbitrary: M rule: marked-lit-list-induct)
  case nil
  thus ?case by (auto simp add: lits-of-def)
next
  case (marked K m) note notin = this(2)
  thus ?case by (auto simp add: lits-of-def)
next
  case (proped L l) note IH = this(1) and L = this(2) and neq = this(3)

```

**show** *?case* **using** *IH[of M@[Propagated L l]] L neq* **by** (*auto simp add: atm-of-eq-atm-of*)  
**qed**

**end**

**theory** *CDCL-W*

**imports** *Partial-Annotated-Clausal-Logic List-More CDCL-W-Level Wellfounded-More*

**begin**

**declare** *set-mset-minus-replicate-mset[simp]*

**lemma** *Bex-set-set-Bex-set[iff]*:  $(\exists x \in \text{set-mset } C. P) \longleftrightarrow (\exists x \in \#C. P)$

**by** *auto*

## 17 Weidenbach's CDCL

**sledgehammer-params**[*verbose, e spass cvc4 z3 verit*]

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

**datatype** *'a conflicting-clause* = *C-True* | *C-Clause 'a*

### 17.1 The State

**locale** *state<sub>W</sub>* =

**fixes**

*trail* :: *'st*  $\Rightarrow$  (*'v, nat, 'v clause*) *marked-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 conflicting-clause* **and**

*cons-trail* :: (*'v, nat, 'v clause*) *marked-lit*  $\Rightarrow$  *'st*  $\Rightarrow$  *'st* **and**

*tl-trail* :: *'st*  $\Rightarrow$  *'st* **and**

*add-init-cls* :: *'v clause*  $\Rightarrow$  *'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 conflicting-clause*  $\Rightarrow$  *'st*  $\Rightarrow$  *'st* **and**

*init-state* :: *'v clauses*  $\Rightarrow$  *'st* **and**

*restart-state* :: *'st*  $\Rightarrow$  *'st*

**assumes**

*trail-cons-trail[simp]*:

$\bigwedge L \text{ st. } \text{undefined-lit } (\text{trail st}) (\text{lit-of } L) \Longrightarrow \text{trail } (\text{cons-trail } L \text{ st}) = L \# \text{trail st}$  **and**

*trail-tl-trail[simp]*:  $\bigwedge \text{st. trail } (\text{tl-trail st}) = \text{tl } (\text{trail st})$  **and**

*trail-add-init-cls[simp]*:

$\bigwedge \text{st } C. \text{no-dup } (\text{trail st}) \Longrightarrow \text{trail } (\text{add-init-cls } C \text{ st}) = \text{trail st}$  **and**

*trail-add-learned-cls[simp]*:

$\bigwedge C \text{ st. no-dup } (\text{trail st}) \Longrightarrow \text{trail } (\text{add-learned-cls } C \text{ st}) = \text{trail st}$  **and**

*trail-remove-cls[simp]*:

$\bigwedge C \text{ st. trail } (\text{remove-cls } C \text{ st}) = \text{trail st}$  **and**

*trail-update-backtrack-lvl[simp]*:  $\bigwedge C \text{ st. trail } (\text{update-backtrack-lvl } C \text{ st}) = \text{trail st}$  **and**

*trail-update-conflicting[simp]*:  $\bigwedge C \text{ st. trail } (\text{update-conflicting } C \text{ st}) = \text{trail st}$  **and**

*init-clss-cons-trail[simp]*:



$\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies init\text{-clss } (cons\text{-trail } M \text{ st}) = init\text{-clss } st \text{ and }$   
 $init\text{-clss-tl-trail}[simp]:$   
 $\bigwedge st. init\text{-clss } (tl\text{-trail } st) = init\text{-clss } st \text{ and }$   
 $init\text{-clss-add-init-cls}[simp]:$   
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies init\text{-clss } (add\text{-init-cls } C \text{ st}) = \{\#C\# \} + init\text{-clss } st \text{ and }$   
 $init\text{-clss-add-learned-cls}[simp]:$   
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies init\text{-clss } (add\text{-learned-cls } C \text{ st}) = init\text{-clss } st \text{ and }$   
 $init\text{-clss-remove-cls}[simp]:$   
 $\bigwedge C \text{ st. init-clss } (remove\text{-cls } C \text{ st}) = remove\text{-mset } C \ (init\text{-clss } st) \text{ and }$   
 $init\text{-clss-update-backtrack-lvl}[simp]:$   
 $\bigwedge st \ C. init\text{-clss } (update\text{-backtrack-lvl } C \text{ st}) = init\text{-clss } st \text{ and }$   
 $init\text{-clss-update-conflicting}[simp]:$   
 $\bigwedge C \text{ st. init-clss } (update\text{-conflicting } C \text{ st}) = init\text{-clss } st \text{ and }$

$learned\text{-clss-cons-trail}[simp]:$   
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$   
 $learned\text{-clss } (cons\text{-trail } M \text{ st}) = learned\text{-clss } st \text{ and }$   
 $learned\text{-clss-tl-trail}[simp]:$   
 $\bigwedge st. learned\text{-clss } (tl\text{-trail } st) = learned\text{-clss } st \text{ and }$   
 $learned\text{-clss-add-init-cls}[simp]:$   
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies learned\text{-clss } (add\text{-init-cls } C \text{ st}) = learned\text{-clss } st \text{ and }$   
 $learned\text{-clss-add-learned-cls}[simp]:$   
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies learned\text{-clss } (add\text{-learned-cls } C \text{ st}) = \{\#C\# \} + learned\text{-clss } st$   
 $\text{and}$   
 $learned\text{-clss-remove-cls}[simp]:$   
 $\bigwedge C \text{ st. learned-clss } (remove\text{-cls } C \text{ st}) = remove\text{-mset } C \ (learned\text{-clss } st) \text{ and }$   
 $learned\text{-clss-update-backtrack-lvl}[simp]:$   
 $\bigwedge st \ C. learned\text{-clss } (update\text{-backtrack-lvl } C \text{ st}) = learned\text{-clss } st \text{ and }$   
 $learned\text{-clss-update-conflicting}[simp]:$   
 $\bigwedge C \text{ st. learned-clss } (update\text{-conflicting } C \text{ st}) = learned\text{-clss } st \text{ and }$

$backtrack\text{-lvl-cons-trail}[simp]:$   
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$   
 $backtrack\text{-lvl } (cons\text{-trail } M \text{ st}) = backtrack\text{-lvl } st \text{ and }$   
 $backtrack\text{-lvl-tl-trail}[simp]:$   
 $\bigwedge st. backtrack\text{-lvl } (tl\text{-trail } st) = backtrack\text{-lvl } st \text{ and }$   
 $backtrack\text{-lvl-add-init-cls}[simp]:$   
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies backtrack\text{-lvl } (add\text{-init-cls } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$   
 $backtrack\text{-lvl-add-learned-cls}[simp]:$   
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies backtrack\text{-lvl } (add\text{-learned-cls } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$   
 $backtrack\text{-lvl-remove-cls}[simp]:$   
 $\bigwedge C \text{ st. backtrack-lvl } (remove\text{-cls } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$   
 $backtrack\text{-lvl-update-backtrack-lvl}[simp]:$   
 $\bigwedge st \ k. backtrack\text{-lvl } (update\text{-backtrack-lvl } k \text{ st}) = k \text{ and }$   
 $backtrack\text{-lvl-update-conflicting}[simp]:$   
 $\bigwedge C \text{ st. backtrack-lvl } (update\text{-conflicting } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$

$conflicting\text{-cons-trail}[simp]:$   
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$   
 $conflicting \ (cons\text{-trail } M \text{ st}) = conflicting \ st \text{ and }$   
 $conflicting\text{-tl-trail}[simp]:$   
 $\bigwedge st. conflicting \ (tl\text{-trail } st) = conflicting \ st \text{ and }$   
 $conflicting\text{-add-init-cls}[simp]:$   
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies conflicting \ (add\text{-init-cls } C \text{ st}) = conflicting \ st \text{ and }$   
 $conflicting\text{-add-learned-cls}[simp]:$

$\bigwedge C \text{ st. no-dup } (\text{trail } st) \implies \text{conflicting } (\text{add-learned-cls } C \text{ st}) = \text{conflicting } st \text{ and}$   
 $\text{conflicting-remove-cls}[simp]:$

$\bigwedge C \text{ st. conflicting } (\text{remove-cls } C \text{ st}) = \text{conflicting } st \text{ and}$   
 $\text{conflicting-update-backtrack-lvl}[simp]:$

$\bigwedge st \ C. \text{ conflicting } (\text{update-backtrack-lvl } C \text{ st}) = \text{conflicting } st \text{ and}$   
 $\text{conflicting-update-conflicting}[simp]:$

$\bigwedge C \text{ st. conflicting } (\text{update-conflicting } C \text{ st}) = C \text{ and}$

$\text{init-state-trail}[simp]: \bigwedge N. \text{ trail } (\text{init-state } N) = [] \text{ and}$

$\text{init-state-clss}[simp]: \bigwedge N. \text{ init-clss } (\text{init-state } N) = N \text{ and}$

$\text{init-state-learned-clss}[simp]: \bigwedge N. \text{ learned-clss } (\text{init-state } N) = \{\#\} \text{ and}$

$\text{init-state-backtrack-lvl}[simp]: \bigwedge N. \text{ backtrack-lvl } (\text{init-state } N) = 0 \text{ and}$

$\text{init-state-conflicting}[simp]: \bigwedge N. \text{ conflicting } (\text{init-state } N) = C\text{-True and}$

$\text{trail-restart-state}[simp]: \text{trail } (\text{restart-state } S) = [] \text{ and}$

$\text{init-clss-restart-state}[simp]: \text{init-clss } (\text{restart-state } S) = \text{init-clss } S \text{ and}$

$\text{learned-clss-restart-state}[intro]: \text{learned-clss } (\text{restart-state } S) \subseteq \# \text{ learned-clss } S \text{ and}$

$\text{backtrack-lvl-restart-state}[simp]: \text{backtrack-lvl } (\text{restart-state } S) = 0 \text{ and}$

$\text{conflicting-restart-state}[simp]: \text{conflicting } (\text{restart-state } S) = C\text{-True}$

**begin**

**definition**  $\text{clauses} :: 'st \Rightarrow 'v \text{ clauses}$  **where**

$\text{clauses } S = \text{init-clss } S + \text{learned-clss } S$

**lemma**

**shows**

$\text{clauses-cons-trail}[simp]:$

$\text{undefined-lit } (\text{trail } S) (\text{lit-of } M) \implies \text{clauses } (\text{cons-trail } M \ S) = \text{clauses } S \text{ and}$

$\text{clauses-tl-trail}[simp]: \text{clauses } (\text{tl-trail } S) = \text{clauses } S \text{ and}$

$\text{clauses-add-learned-clss-unfolded}: \text{no-dup } (\text{trail } S) \implies$

$\text{clauses } (\text{add-learned-clss } U \ S) = \{\#U\# \} + \text{learned-clss } S + \text{init-clss } S$   
**and**

$\text{clauses-add-init-clss}[simp]:$

$\text{no-dup } (\text{trail } S) \implies \text{clauses } (\text{add-init-clss } N \ S) = \{\#N\# \} + \text{init-clss } S + \text{learned-clss } S \text{ and}$

$\text{clauses-update-backtrack-lvl}[simp]: \text{clauses } (\text{update-backtrack-lvl } k \ S) = \text{clauses } S \text{ and}$

$\text{clauses-update-conflicting}[simp]: \text{clauses } (\text{update-conflicting } D \ S) = \text{clauses } S \text{ and}$

$\text{clauses-remove-clss}[simp]:$

$\text{clauses } (\text{remove-clss } C \ S) = \text{clauses } S - \text{replicate-mset } (\text{count } (\text{clauses } S) \ C) \ C \text{ and}$

$\text{clauses-add-learned-clss}[simp]:$

$\text{no-dup } (\text{trail } S) \implies \text{clauses } (\text{add-learned-clss } C \ S) = \{\#C\# \} + \text{clauses } S \text{ and}$

$\text{clauses-restart}[simp]: \text{clauses } (\text{restart-state } S) \subseteq \# \text{ clauses } S \text{ and}$

$\text{clauses-init-state}[simp]: \bigwedge N. \text{ clauses } (\text{init-state } N) = N$

**prefer 9 using**  $\text{clauses-def learned-clss-restart-state}$  **apply**  $\text{fastforce}$

**by**  $(\text{auto simp: ac-simps replicate-mset-plus clauses-def intro: multiset-eqI})$

**abbreviation**  $\text{state} :: 'st \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit list} \times 'v \text{ clauses} \times 'v \text{ clauses}$

$\times \text{nat} \times 'v \text{ clause conflicting-clause}$  **where**

$\text{state } S \equiv (\text{trail } S, \text{init-clss } S, \text{learned-clss } S, \text{backtrack-lvl } S, \text{conflicting } S)$

**abbreviation**  $\text{incr-lvl} :: 'st \Rightarrow 'st$  **where**

$\text{incr-lvl } S \equiv \text{update-backtrack-lvl } (\text{backtrack-lvl } S + 1) \ S$

**definition**  $\text{state-eq} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  **(infix  $\sim$  50)** **where**

$S \sim T \iff \text{state } S = \text{state } T$

**lemma** *state-eq-ref*[*simp*, *intro*]:  
 $S \sim S$   
**unfolding** *state-eq-def* **by** *auto*

**lemma** *state-eq-sym*:  
 $S \sim T \iff T \sim S$   
**unfolding** *state-eq-def* **by** *auto*

**lemma** *state-eq-trans*:  
 $S \sim T \implies T \sim U \implies S \sim U$   
**unfolding** *state-eq-def* **by** *auto*

**lemma**  
**shows**  
*state-eq-trail*:  $S \sim T \implies \text{trail } S = \text{trail } T$  **and**  
*state-eq-init-clss*:  $S \sim T \implies \text{init-clss } S = \text{init-clss } T$  **and**  
*state-eq-learned-clss*:  $S \sim T \implies \text{learned-clss } S = \text{learned-clss } T$  **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*:  $S \sim T \implies \text{undefined-lit } (\text{trail } S) L = \text{undefined-lit } (\text{trail } T) L$   
**unfolding** *state-eq-def* *clauses-def* **by** *auto*

**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*

**lemma** *atms-of-m-learned-clss-restart-state-in-atms-of-m-learned-clssI*[*intro*]:  
 $x \in \text{atms-of-mu } (\text{learned-clss } (\text{restart-state } S)) \implies x \in \text{atms-of-mu } (\text{learned-clss } S)$   
**by** (*meson* *atms-of-m-mono* *learned-clss-restart-state* *set-mset-mono* *subsetCE*)

**function** *reduce-trail-to* :: (*'v*, *nat*, *'v clause*) *marked-lits*  $\Rightarrow$  *'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*))  
**by** *fast+*  
**termination**  
**by** (*relation measure* ( $\lambda(-, S). \text{length } (\text{trail } S)$ )) *simp-all*

**declare** *reduce-trail-to.simps*[*simp* *del*]

**lemma**  
**shows**  
*reduce-trail-to-nil*[*simp*]:  $\text{trail } S = [] \implies \text{reduce-trail-to } F S = S$  **and**  
*reduce-trail-to-eq-length*[*simp*]:  $\text{length } (\text{trail } S) = \text{length } F \implies \text{reduce-trail-to } F S = S$   
**by** (*auto* *simp*: *reduce-trail-to.simps*)

**lemma** *reduce-trail-to-length-ne*:  
 $\text{length } (\text{trail } S) \neq \text{length } F \implies \text{trail } S \neq [] \implies$   
 $\text{reduce-trail-to } F S = \text{reduce-trail-to } F (\text{tl-trail } S)$   
**by** (*auto* *simp*: *reduce-trail-to.simps*)

**lemma** *trail-reduce-trail-to-length-le*:  
**assumes**  $\text{length } F > \text{length } (\text{trail } S)$   
**shows**  $\text{trail } (\text{reduce-trail-to } F S) = []$   
**using** *assms* **apply** (*induction* *F* *S* *rule*: *reduce-trail-to.induct*)

**by** (*metis* (*no-types*, *hide-lams*) *length-tl less-imp-diff-less less-irrefl trail-tl-trail reduce-trail-to.simps*)

**lemma** *trail-reduce-trail-to-nil[simp]*:

*trail* (*reduce-trail-to* [] *S*) = []

**apply** (*induction* []:: ('v, nat, 'v clause) *marked-lits S rule: reduce-trail-to.induct*)

**by** (*metis* *length-0-conv reduce-trail-to-length-ne reduce-trail-to-nil*)

**lemma** *clauses-reduce-trail-to-nil*:

*clauses* (*reduce-trail-to* [] *S*) = *clauses S*

**apply** (*induction* []:: ('v, nat, 'v clause) *marked-lits S rule: reduce-trail-to.induct*)

**by** (*metis* *clauses-tl-trail reduce-trail-to.simps*)

**lemma** *reduce-trail-to-skip-beginning*:

**assumes** *trail S = F' @ F*

**shows** *trail (reduce-trail-to F S) = F*

**using** *assms* **by** (*induction F' arbitrary: S*) (*auto simp: reduce-trail-to-length-ne*)

**lemma** *clauses-reduce-trail-to[simp]*:

*clauses* (*reduce-trail-to F S*) = *clauses S*

**apply** (*induction F S rule: reduce-trail-to.induct*)

**by** (*metis* *clauses-tl-trail reduce-trail-to.simps*)

**lemma** *conflicting-update-trial[simp]*:

*conflicting* (*reduce-trail-to F S*) = *conflicting S*

**apply** (*induction F S rule: reduce-trail-to.induct*)

**by** (*metis* *conflicting-tl-trail reduce-trail-to.simps*)

**lemma** *backtrack-lvl-update-trial[simp]*:

*backtrack-lvl* (*reduce-trail-to F S*) = *backtrack-lvl S*

**apply** (*induction F S rule: reduce-trail-to.induct*)

**by** (*metis* *backtrack-lvl-tl-trail reduce-trail-to.simps*)

**lemma** *init-clss-update-trial[simp]*:

*init-clss* (*reduce-trail-to F S*) = *init-clss S*

**apply** (*induction F S rule: reduce-trail-to.induct*)

**by** (*metis* *init-clss-tl-trail reduce-trail-to.simps*)

**lemma** *learned-clss-update-trial[simp]*:

*learned-clss* (*reduce-trail-to F S*) = *learned-clss S*

**apply** (*induction F S rule: reduce-trail-to.induct*)

**by** (*metis* *learned-clss-tl-trail reduce-trail-to.simps*)

**lemma** *trail-eq-reduce-trail-to-eq*:

*trail S = trail T*  $\implies$  *trail (reduce-trail-to F S) = trail (reduce-trail-to F T)*

**apply** (*induction F S arbitrary: T rule: reduce-trail-to.induct*)

**by** (*metis* *trail-tl-trail reduce-trail-to.simps*)

**lemma** *reduce-trail-to-state-eq<sub>NOT</sub>-compatible*:

**assumes** *ST: S ~ T*

**shows** *reduce-trail-to F S ~ reduce-trail-to F T*

**proof** –

**have** *trail (reduce-trail-to F S) = trail (reduce-trail-to F T)*

**using** *trail-eq-reduce-trail-to-eq[of S T F] ST* **by** *auto*

**then show** *?thesis* **using** *ST* **by** (*auto simp del: state-simp simp: state-eq-def*)

qed

**lemma** *reduce-trail-to-trail-tl-trail-decomp[simp]:*

*trail S = F' @ Marked K d # F  $\implies$  (trail (reduce-trail-to F S)) = F*  
**apply** (rule *reduce-trail-to-skip-beginning*[of - F' @ Marked K d # []])  
**by** (cases F') (auto simp add:tl-append reduce-trail-to-skip-beginning)

**lemma** *reduce-trail-to-add-learned-cls[simp]:*

*no-dup (trail S)  $\implies$*   
*trail (reduce-trail-to F (add-learned-cls C S)) = trail (reduce-trail-to F S)*  
**by** (rule *trail-eq-reduce-trail-to-eq*) auto

**lemma** *reduce-trail-to-add-init-cls[simp]:*

*no-dup (trail S)  $\implies$*   
*trail (reduce-trail-to F (add-init-cls C S)) = trail (reduce-trail-to F S)*  
**by** (rule *trail-eq-reduce-trail-to-eq*) auto

**lemma** *reduce-trail-to-remove-learned-cls[simp]:*

*trail (reduce-trail-to F (remove-cls C S)) = trail (reduce-trail-to F S)*  
**by** (rule *trail-eq-reduce-trail-to-eq*) auto

**lemma** *reduce-trail-to-update-conflicting[simp]:*

*trail (reduce-trail-to F (update-conflicting C S)) = trail (reduce-trail-to F S)*  
**by** (rule *trail-eq-reduce-trail-to-eq*) auto

**lemma** *reduce-trail-to-update-backtrack-lvl[simp]:*

*trail (reduce-trail-to F (update-backtrack-lvl C S)) = trail (reduce-trail-to F S)*  
**by** (rule *trail-eq-reduce-trail-to-eq*) auto

**lemma** *in-get-all-marked-decomposition-marked-or-empty:*

**assumes**  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$   
**shows**  $a = [] \vee (\text{is-marked } (\text{hd } a))$   
**using** *assms*

**proof** (induct M arbitrary: a b)

**case** Nil **then show** ?case **by** simp

**next**

**case** (Cons m M)

**show** ?case

**proof** (cases m)

**case** (Marked l mark)

**then show** ?thesis **using** Cons **by** auto

**next**

**case** (Propagated l mark)

**then show** ?thesis **using** Cons **by** (cases get-all-marked-decomposition M) force+

qed

qed

**lemma** *in-get-all-marked-decomposition-trail-update-trail[simp]:*

**assumes**  $H: (L \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$   
**shows**  $\text{trail } (\text{reduce-trail-to } M1 \ S) = M1$

**proof** –

**obtain** K mark **where**

$L: L = \text{Marked } K \ \text{mark}$

**using** H **by** (cases L) (auto dest!: in-get-all-marked-decomposition-marked-or-empty)

**obtain** c **where**

```

  tr-S: trail S = c @ M2 @ L # M1
  using H by auto
  show ?thesis
  by (rule reduce-trail-to-trail-tl-trail-decomp[of - c @ M2 K mark])
  (auto simp: tr-S L)
qed

```

```

fun append-trail where
  append-trail [] S = S |
  append-trail (L # M) S = append-trail M (cons-trail L S)

```

```

lemma trail-append-trail[simp]:
  no-dup (M @ trail S)  $\implies$  trail (append-trail M S) = rev M @ trail S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

```

lemma learned-clss-append-trail[simp]:
  no-dup (M @ trail S)  $\implies$  learned-clss (append-trail M S) = learned-clss S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

```

lemma init-clss-append-trail[simp]:
  no-dup (M @ trail S)  $\implies$  init-clss (append-trail M S) = init-clss S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

```

lemma conflicting-append-trail[simp]:
  no-dup (M @ trail S)  $\implies$  conflicting (append-trail M S) = conflicting S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

```

lemma backtrack-lvl-append-trail[simp]:
  no-dup (M @ trail S)  $\implies$  backtrack-lvl (append-trail M S) = backtrack-lvl S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

```

lemma clauses-append-trail[simp]:
  no-dup (M @ trail S)  $\implies$  clauses (append-trail M S) = clauses S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

This function is useful for proofs to speak of a global trail change, but is a bad for programs and code in general.

```

fun delete-trail-and-rebuild where
  delete-trail-and-rebuild M S = append-trail (rev M) (reduce-trail-to [] S)

```

**end**

## 17.2 Special Instantiation: using Triples as State

### 17.3 CDCL Rules

Because of the strategy we will later use, we distinguish propagate, conflict from the other rules

```

locale
  cdclW-ops =
    stateW trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail add-init-clss
    add-learned-clss remove-clss update-backtrack-lvl update-conflicting init-state
    restart-state
for
  trail :: 'st  $\Rightarrow$  ('v, nat, 'v clause) marked-lits and
  init-clss :: 'st  $\Rightarrow$  'v clauses and

```

```

learned-cls :: 'st  $\Rightarrow$  'v clauses and
backtrack-lvl :: 'st  $\Rightarrow$  nat and
conflicting :: 'st  $\Rightarrow$  'v clause conflicting-clause and

cons-trail :: ('v, nat, 'v clause) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-init-cls :: 'v clause  $\Rightarrow$  '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 conflicting-clause  $\Rightarrow$  'st  $\Rightarrow$  'st and

init-state :: 'v clauses  $\Rightarrow$  'st and
restart-state :: 'st  $\Rightarrow$  'st
begin

inductive propagate :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
propagate-rule[intro]:
  state  $S = (M, N, U, k, C\text{-True}) \Rightarrow C + \{\#L\# \} \in \# \text{ clauses } S \Rightarrow M \models_{as} C\text{Not } C$ 
 $\Rightarrow$  undefined-lit (trail  $S$ )  $L$ 
 $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S$ 
 $\Rightarrow \text{propagate } S T$ 
inductive-cases propagateE[elim]: propagate  $S T$ 
thm propagateE

inductive conflict :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
conflict-rule[intro]: state  $S = (M, N, U, k, C\text{-True}) \Rightarrow D \in \# \text{ clauses } S \Rightarrow M \models_{as} C\text{Not } D$ 
 $\Rightarrow T \sim \text{update-conflicting } (C\text{-Clause } D) S$ 
 $\Rightarrow \text{conflict } S T$ 

inductive-cases conflictE[elim]: conflict  $S S'$ 

inductive backtrack :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
backtrack-rule[intro]: state  $S = (M, N, U, k, C\text{-Clause } (D + \{\#L\# \}))$ 
 $\Rightarrow (\text{Marked } K (i+1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
 $\Rightarrow \text{get-level } L M = k$ 
 $\Rightarrow \text{get-level } L M = \text{get-maximum-level } (D + \{\#L\# \}) M$ 
 $\Rightarrow \text{get-maximum-level } D M = i$ 
 $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$ 
  (reduce-trail-to  $M1$ 
    (add-learned-cls  $(D + \{\#L\# \})$ 
      (update-backtrack-lvl  $i$ 
        (update-conflicting  $C\text{-True } S$ ))))
 $\Rightarrow \text{backtrack } S T$ 
inductive-cases backtrackE[elim]: backtrack  $S S'$ 
thm backtrackE

inductive decide :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
decide-rule[intro]: state  $S = (M, N, U, k, C\text{-True})$ 
 $\Rightarrow \text{undefined-lit } M L \Rightarrow \text{atm-of } L \in \text{atms-of-mu } (\text{init-cls } S)$ 
 $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L (k+1)) (\text{incr-lvl } S)$ 
 $\Rightarrow \text{decide } S T$ 
inductive-cases decideE[elim]: decide  $S S'$ 
thm decideE

```

**inductive** *skip* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **where**  
*skip-rule*[intro]: state  $S = (\text{Propagated } L \ C' \# M, N, U, k, \text{C-Clause } D) \Longrightarrow -L \notin \# D \Longrightarrow D \neq \{\#\}$   
 $\Longrightarrow T \sim \text{tl-trail } S$   
 $\Longrightarrow \text{skip } S \ T$   
**inductive-cases** *skipE*[elim]: *skip*  $S \ S'$   
**thm** *skipE*

*get-maximum-level*  $D \ (\text{Propagated } L \ (C + \{\#L\#\}) \# M) = k \vee k = 0$  is equivalent to  
*get-maximum-level*  $D \ (\text{Propagated } L \ (C + \{\#L\#\}) \# M) = k$

**inductive** *resolve* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **where**  
*resolve-rule*[intro]:  
state  $S = (\text{Propagated } L \ ((C + \{\#L\#\})) \# M, N, U, k, \text{C-Clause } (D + \{\#-L\#\}))$   
 $\Longrightarrow \text{get-maximum-level } D \ (\text{Propagated } L \ (C + \{\#L\#\}) \# M) = k$   
 $\Longrightarrow T \sim \text{update-conflicting } (\text{C-Clause } (D \# \cup C)) \ (\text{tl-trail } S)$   
 $\Longrightarrow \text{resolve } S \ T$   
**inductive-cases** *resolveE*[elim]: *resolve*  $S \ S'$   
**thm** *resolveE*

**inductive** *restart* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **where**  
*restart*: state  $S = (M, N, U, k, \text{C-True}) \Longrightarrow \neg M \models_{\text{asm}} \text{clauses } S$   
 $\Longrightarrow T \sim \text{restart-state } S$   
 $\Longrightarrow \text{restart } S \ T$   
**inductive-cases** *restartE*[elim]: *restart*  $S \ T$   
**thm** *restartE*

We add the condition  $C \notin \# \text{init-clss } S$ , to maintain consistency even without the strategy.

**inductive** *forget* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **where**  
*forget-rule*: state  $S = (M, N, \{\#C\#\} + U, k, \text{C-True})$   
 $\Longrightarrow \neg M \models_{\text{asm}} \text{clauses } S$   
 $\Longrightarrow C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$   
 $\Longrightarrow C \notin \# \text{init-clss } S$   
 $\Longrightarrow C \in \# \text{learned-clss } S$   
 $\Longrightarrow T \sim \text{remove-cl } C \ S$   
 $\Longrightarrow \text{forget } S \ T$   
**inductive-cases** *forgetE*[elim]: *forget*  $S \ T$

**inductive** *cdcl<sub>W</sub>-rf* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for**  $S :: 'st$  **where**  
*restart*: *restart*  $S \ T \Longrightarrow \text{cdcl}_W\text{-rf } S \ T \mid$   
*forget*: *forget*  $S \ T \Longrightarrow \text{cdcl}_W\text{-rf } S \ T$

**inductive** *cdcl<sub>W</sub>-bj* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **where**  
*skip*[intro]: *skip*  $S \ S' \Longrightarrow \text{cdcl}_W\text{-bj } S \ S' \mid$   
*resolve*[intro]: *resolve*  $S \ S' \Longrightarrow \text{cdcl}_W\text{-bj } S \ S' \mid$   
*backtrack*[intro]: *backtrack*  $S \ S' \Longrightarrow \text{cdcl}_W\text{-bj } S \ S'$

**inductive-cases** *cdcl<sub>W</sub>-bjE*: *cdcl<sub>W</sub>-bj*  $S \ T$

**inductive** *cdcl<sub>W</sub>-o*: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for**  $S :: 'st$  **where**  
*decide*[intro]: *decide*  $S \ S' \Longrightarrow \text{cdcl}_W\text{-o } S \ S' \mid$   
*bj*[intro]: *cdcl<sub>W</sub>-bj*  $S \ S' \Longrightarrow \text{cdcl}_W\text{-o } S \ S'$

**inductive** *cdcl<sub>W</sub>* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for**  $S :: 'st$  **where**  
*propagate*: *propagate*  $S \ S' \Longrightarrow \text{cdcl}_W \ S \ S' \mid$   
*conflict*: *conflict*  $S \ S' \Longrightarrow \text{cdcl}_W \ S \ S' \mid$   
*other*: *cdcl<sub>W</sub>-o*  $S \ S' \Longrightarrow \text{cdcl}_W \ S \ S' \mid$



*rf*:  $cdcl_W\text{-rf } S \ S' \implies cdcl_W \ S \ S'$

**lemma** *rtrancpl-propagate-is-rtrancpl-cdcl<sub>W</sub>*:  
 $propagate^{**} \ S \ S' \implies cdcl_W^{**} \ S \ S'$   
**by** (*induction rule*: *rtrancpl-induct*) (*fastforce dest*!: *propagate*)+

**lemma** *cdcl<sub>W</sub>-all-rules-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

**fixes** *S* :: 'st  
**assumes**  
 $cdcl_W: cdcl_W \ S \ S' \text{ and}$   
 $propagate: \bigwedge T. propagate \ S \ T \implies P \ S \ T \text{ and}$   
 $conflict: \bigwedge T. conflict \ S \ T \implies P \ S \ T \text{ and}$   
 $forget: \bigwedge T. forget \ S \ T \implies P \ S \ T \text{ and}$   
 $restart: \bigwedge T. restart \ S \ T \implies P \ S \ T \text{ and}$   
 $decide: \bigwedge T. decide \ S \ T \implies P \ S \ T \text{ and}$   
 $skip: \bigwedge T. skip \ S \ T \implies P \ S \ T \text{ and}$   
 $resolve: \bigwedge T. resolve \ S \ T \implies P \ S \ T \text{ and}$   
 $backtrack: \bigwedge T. backtrack \ S \ T \implies P \ S \ T$   
**shows**  $P \ S \ S'$   
**using** *assms*(1)  
**proof** (*induct S' rule*: *cdcl<sub>W</sub>.induct*)  
**case** (*propagate S'*) **note** *propagate = this*(1)  
**then show** ?*case* **using** *assms*(2) **by** *auto*  
**next**  
**case** (*conflict S'*)  
**then show** ?*case* **using** *assms*(3) **by** *auto*  
**next**  
**case** (*other S'*)  
**then show** ?*case*  
**proof** (*induct rule*: *cdcl<sub>W</sub>-o.induct*)  
**case** (*decide U*)  
**then show** ?*case* **using** *assms*(6) **by** *auto*  
**next**  
**case** (*bj S'*)  
**then show** ?*case* **using** *assms*(7–9) **by** (*induction rule*: *cdcl<sub>W</sub>-bj.induct*) *auto*  
**qed**  
**next**  
**case** (*rf S'*)  
**then show** ?*case*  
**by** (*induct rule*: *cdcl<sub>W</sub>-rf.induct*) (*fast dest*: *forget restart*) +  
**qed**

**lemma** *cdcl<sub>W</sub>-all-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

**fixes** *S* :: 'st  
**assumes**  
 $cdcl_W: cdcl_W \ S \ S' \text{ and}$   
 $propagateH: \bigwedge C \ L \ T. C + \{\#L\# \} \in \# \text{ clauses } S \implies trail \ S \models_{as} C \text{Not } C$   
 $\implies \text{undefined-lit } (trail \ S) \ L \implies \text{conflicting } S = C\text{-True}$   
 $\implies T \sim \text{cons-trail } (Propagated \ L \ (C + \{\#L\# \})) \ S$   
 $\implies P \ S \ T \text{ and}$   
 $conflictH: \bigwedge D \ T. D \in \# \text{ clauses } S \implies \text{conflicting } S = C\text{-True} \implies trail \ S \models_{as} C \text{Not } D$   
 $\implies T \sim \text{update-conflicting } (C\text{-Clause } D) \ S$   
 $\implies P \ S \ T \text{ and}$

$forgetH: \bigwedge C T. \neg trail\ S \models_{asm} clauses\ S$   
 $\implies C \notin set\ (get-all-mark-of-propagated\ (trail\ S))$   
 $\implies C \notin \# init-clss\ S$   
 $\implies C \in \# learned-clss\ S$   
 $\implies conflicting\ S = C-True$   
 $\implies T \sim remove-cl\ C\ S$   
 $\implies P\ S\ T\ \mathbf{and}$   
 $restartH: \bigwedge T. \neg trail\ S \models_{asm} clauses\ S$   
 $\implies conflicting\ S = C-True$   
 $\implies T \sim restart-state\ S$   
 $\implies P\ S\ T\ \mathbf{and}$   
 $decideH: \bigwedge L T. conflicting\ S = C-True \implies undefined-lit\ (trail\ S)\ L$   
 $\implies atm-of\ L \in atms-of-mu\ (init-clss\ S)$   
 $\implies T \sim cons-trail\ (Marked\ L\ (backtrack-lvl\ S + 1))\ (incr-lvl\ S)$   
 $\implies P\ S\ T\ \mathbf{and}$   
 $skipH: \bigwedge L C' M D T. trail\ S = Propagated\ L\ C' \# M$   
 $\implies conflicting\ S = C-Clause\ D \implies -L \notin \# D \implies D \neq \{\#\}$   
 $\implies T \sim tl-trail\ S$   
 $\implies P\ S\ T\ \mathbf{and}$   
 $resolveH: \bigwedge L C M D T.$   
 $trail\ S = Propagated\ L\ ((C + \{\#L\#})) \# M$   
 $\implies conflicting\ S = C-Clause\ (D + \{\#-L\#})$   
 $\implies get-maximum-level\ D\ (Propagated\ L\ ((C + \{\#L\#})) \# M) = backtrack-lvl\ L$   
 $\implies T \sim (update-conflicting\ (C-Clause\ (D \# \cup C))\ (tl-trail\ S))$   
 $\implies P\ S\ T\ \mathbf{and}$   
 $backtrackH: \bigwedge K i M1 M2 L D T.$   
 $(Marked\ K\ (Suc\ i) \# M1, M2) \in set\ (get-all-marked-decomposition\ (trail\ S))$   
 $\implies get-level\ L\ (trail\ S) = backtrack-lvl\ S$   
 $\implies conflicting\ S = C-Clause\ (D + \{\#L\#})$   
 $\implies get-maximum-level\ (D + \{\#L\#})\ (trail\ S) = get-level\ L\ (trail\ S)$   
 $\implies get-maximum-level\ D\ (trail\ S) \equiv i$   
 $\implies T \sim cons-trail\ (Propagated\ L\ (D + \{\#L\#}))$   
 $\quad (reduce-trail-to\ M1$   
 $\quad \quad (add-learned-cl\ (D + \{\#L\#})$   
 $\quad \quad \quad (update-backtrack-lvl\ i$   
 $\quad \quad \quad \quad (update-conflicting\ C-True\ S))))$   
 $\implies P\ S\ T$   
**shows**  $P\ S\ S'$   
**using**  $cdcl_W$   
**proof** (*induct*  $S\ S'$  *rule*:  $cdcl_W$ -all-rules-induct)  
**case** (*propagate*  $S'$ )  
**then show** ?*case by* (*elim propagateE*) (*frule propagateH; simp*)  
**next**  
**case** (*conflict*  $S'$ )  
**then show** ?*case by* (*elim conflictE*) (*frule conflictH; simp*)  
**next**  
**case** (*restart*  $S'$ )  
**then show** ?*case by* (*elim restartE*) (*frule restartH; simp*)  
**next**  
**case** (*decide*  $T$ )  
**then show** ?*case by* (*elim decideE*) (*frule decideH; simp*)  
**next**  
**case** (*backtrack*  $S'$ )  
**then show** ?*case by* (*elim backtrackE*) (*frule backtrackH; simp del: state-simp add: state-eq-def*)  
**next**

```

  case (forget S')
  then show ?case using forgetH by auto
next
  case (skip S')
  then show ?case using skipH by auto
next
  case (resolve S')
  then show ?case by (elim resolveE) (frule resolveH; simp)
qed

```

**lemma** *cdcl<sub>W</sub>-o-induct*[consumes 1, case-names decide skip resolve backtrack]:  
**fixes**  $S :: 'st$   
**assumes** *cdcl<sub>W</sub>*: *cdcl<sub>W</sub>-o*  $S$   $T$  **and**  
*decideH*:  $\bigwedge L T. \text{conflicting } S = C\text{-True} \implies \text{undefined-lit } (\text{trail } S) L$   
 $\implies \text{atm-of } L \in \text{atms-of-mu } (\text{init-clss } S)$   
 $\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$   
 $\implies P S T$  **and**  
*skipH*:  $\bigwedge L C' M D T. \text{trail } S = \text{Propagated } L C' \# M$   
 $\implies \text{conflicting } S = C\text{-Clause } D \implies -L \notin \# D \implies D \neq \{\#\}$   
 $\implies T \sim \text{tl-trail } S$   
 $\implies P S T$  **and**  
*resolveH*:  $\bigwedge L C M D T.$   
 $\text{trail } S = \text{Propagated } L ( (C + \{\#L\# \}) \# M$   
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#-L\# \})$   
 $\implies \text{get-maximum-level } D (\text{Propagated } L (C + \{\#L\# \}) \# M) = \text{backtrack-lvl } S$   
 $\implies T \sim \text{update-conflicting } (C\text{-Clause } (D \# \cup C)) (\text{tl-trail } S)$   
 $\implies P S T$  **and**  
*backtrackH*:  $\bigwedge K i M1 M2 L D T.$   
 $(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$   
 $\implies \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S$   
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$   
 $\implies \text{get-level } L (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$   
 $\implies \text{get-maximum-level } D (\text{trail } S) \equiv i$   
 $\implies T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$   
 $\quad (\text{reduce-trail-to } M1$   
 $\quad \quad (\text{add-learned-cls } (D + \{\#L\# \})$   
 $\quad \quad \quad (\text{update-backtrack-lvl } i$   
 $\quad \quad \quad \quad (\text{update-conflicting } C\text{-True } S))))$   
 $\implies P S T$   
**shows**  $P S T$   
**using** *cdcl<sub>W</sub>* **apply** (*induct*  $T$  *rule*: *cdcl<sub>W</sub>-o.induct*)  
**using** *assms*(2) **apply** *auto*[1]  
**apply** (*elim* *cdcl<sub>W</sub>-bjE* *skipE* *resolveE* *backtrackE*)  
**apply** (*frule* *skipH*; *simp*)  
**apply** (*frule* *resolveH*; *simp*)  
**apply** (*frule* *backtrackH*; *simp-all* *del*: *state-simp* *add*: *state-eq-def*)  
**done**

**thm** *cdcl<sub>W</sub>-o.induct*

**lemma** *cdcl<sub>W</sub>-o-all-rules-induct*[consumes 1, case-names decide backtrack skip resolve]:  
**fixes**  $S T :: 'st$   
**assumes**  
*cdcl<sub>W</sub>-o*  $S T$  **and**  
 $\bigwedge T. \text{decide } S T \implies P S T$  **and**  
 $\bigwedge T. \text{backtrack } S T \implies P S T$  **and**

$\bigwedge T. \text{skip } S \ T \implies P \ S \ T$  **and**  
 $\bigwedge T. \text{resolve } S \ T \implies P \ S \ T$   
**shows**  $P \ S \ T$   
**using** *assms* **by** (*induct*  $T$  *rule*:  $\text{cdcl}_W\text{-o.induct}$ ) (*auto simp*:  $\text{cdcl}_W\text{-bj.simps}$ )

**lemma**  $\text{cdcl}_W\text{-o-rule-cases}$ [*consumes* 1, *case-names* *decide backtrack skip resolve*]:  
**fixes**  $S \ T :: 'st$   
**assumes**  
 $\text{cdcl}_W\text{-o } S \ T$  **and**  
 $\text{decide } S \ T \implies P$  **and**  
 $\text{backtrack } S \ T \implies P$  **and**  
 $\text{skip } S \ T \implies P$  **and**  
 $\text{resolve } S \ T \implies P$   
**shows**  $P$   
**using** *assms* **by** (*auto simp*:  $\text{cdcl}_W\text{-o.simps}$   $\text{cdcl}_W\text{-bj.simps}$ )

## 17.4 Invariants

### 17.4.1 Properties of the trail

We here establish that: \* the marks are exactly 1..k where k is the level \* the consistency of the trail \* the fact that there is no duplicate in the trail.

**lemma** *backtrack-lit-skipped*:  
**assumes**  $L$ :  $\text{get-level } L \ (\text{trail } S) = \text{backtrack-lvl } S$   
**and**  $M1$ :  $(\text{Marked } K \ (i + 1) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$   
**and** *no-dup*:  $\text{no-dup } (\text{trail } S)$   
**and** *bt-l*:  $\text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$   
**and** *order*:  $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } ([1..<(1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S)))])$   
**shows**  $\text{atm-of } L \notin \text{atm-of ' lits-of } M1$   
**proof**  
**let**  $?M = \text{trail } S$   
**assume**  $L\text{-in-}M1$ :  $\text{atm-of } L \in \text{atm-of ' lits-of } M1$   
**obtain**  $c$  **where**  $Mc$ :  $\text{trail } S = c @ M2 @ \text{Marked } K \ (i + 1) \ \# \ M1$  **using**  $M1$  **by** *blast*  
**have**  $\text{atm-of } L \notin \text{atm-of ' lits-of } c$   
**using**  $L\text{-in-}M1$  *no-dup mk-disjoint-insert unfolding*  $Mc$  *lits-of-def* **by** *force*  
**have**  $g\text{-}M\text{-eq-}g\text{-}M1$ :  $\text{get-level } L \ ?M = \text{get-level } L \ M1$   
**using**  $L\text{-in-}M1$  *unfolding*  $Mc$  **by** *auto*  
**have**  $g$ :  $\text{get-all-levels-of-marked } M1 = \text{rev } [1..<\text{Suc } i]$   
**using** *order* *unfolding*  $Mc$   
**by** (*auto simp del*: *upt-simps dest*!: *append-cons-eq-upt-length-i*  
*simp add*: *rev-swap[symmetric]*)  
**then have**  $\text{Max } (\text{set } (0 \ \# \ \text{get-all-levels-of-marked } (\text{rev } M1))) < \text{Suc } i$  **by** *auto*  
**then have**  $\text{get-level } L \ M1 < \text{Suc } i$   
**using**  $\text{get-rev-level-less-max-get-all-levels-of-marked}$ [*of*  $L \ 0 \ \text{rev } M1$ ] **by** *linarith*  
**moreover have**  $\text{Suc } i \leq \text{backtrack-lvl } S$  **using** *bt-l* **by** (*simp add*:  $Mc \ g$ )  
**ultimately show** *False* **using**  $L \ g\text{-}M\text{-eq-}g\text{-}M1$  **by** *auto*  
**qed**

**lemma**  $\text{cdcl}_W\text{-distinctinv-1}$ :  
**assumes**  
 $\text{cdcl}_W \ S \ S'$  **and**  
 $\text{no-dup } (\text{trail } S)$  **and**  
 $\text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$  **and**  
 $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [1..<1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$

**shows** *no-dup* (*trail S'*)  
**using** *assms*  
**proof** (*induct rule: cdcl<sub>W</sub>-all-induct*)  
**case** (*backtrack K i M1 M2 L D T*) **note** *decomp = this(1)* **and** *L = this(2)* **and** *T = this(6)* **and**  
*n-d = this(7)*  
**obtain** *c* **where** *Mc: trail S = c @ M2 @ Marked K (i + 1) # M1*  
**using** *decomp* **by** *auto*  
**have** *no-dup (M2 @ Marked K (i + 1) # M1)*  
**using** *Mc n-d* **by** *fastforce*  
**moreover** **have** *atm-of L ∉ (λl. atm-of (lit-of l))* ‘ *set M1*  
**using** *backtrack-lit-skipped[of L S K i M1 M2] L decomp backtrack.prem*  
**by** (*fastforce simp add: lits-of-def*)  
**moreover** **then** **have** *undefined-lit M1 L*  
**by** (*simp add: defined-lit-map*)  
**ultimately** **show** *?case* **using** *decomp T n-d* **by** *simp*  
**qed** (*auto simp add: defined-lit-map*)

**lemma** *cdcl<sub>W</sub>-consistent-inv-2:*

**assumes**  
*cdcl<sub>W</sub> S S' and*  
*no-dup (trail S) and*  
*backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and*  
*get-all-levels-of-marked (trail S) = rev [1..*1+length (get-all-levels-of-marked (trail S))*]*  
**shows** *consistent-interp (lits-of (trail S'))*  
**using** *cdcl<sub>W</sub>-distinctinv-1[OF assms] distinctconsistent-interp* **by** *fast*

**lemma** *cdcl<sub>W</sub>-o-bt:*

**assumes**  
*cdcl<sub>W</sub>-o S S' and*  
*backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and*  
*get-all-levels-of-marked (trail S) =*  
*rev ([1..*1+length (get-all-levels-of-marked (trail S))*])) and*  
*n-d[simp]: no-dup (trail S)*  
**shows** *backtrack-lvl S' = length (get-all-levels-of-marked (trail S'))*  
**using** *assms*  
**proof** (*induct rule: cdcl<sub>W</sub>-o-induct*)  
**case** (*backtrack K i M1 M2 L D T*) **note** *decomp = this(1)* **and** *T = this(6)* **and** *level = this(8)*  
**have** [*simp*]: *trail (reduce-trail-to M1 S) = M1*  
**using** *decomp* **by** *auto*  
**obtain** *c* **where** *M: trail S = c @ M2 @ Marked K (i + 1) # M1* **using** *decomp* **by** *auto*  
**have** *rev (get-all-levels-of-marked (trail S))*  
*= [1..*1+length (get-all-levels-of-marked (trail S))*]*  
**using** *level* **by** (*auto simp: rev-swap[symmetric]*)  
**moreover** **have** *atm-of L ∉ (λl. atm-of (lit-of l))* ‘ *set M1*  
**using** *backtrack-lit-skipped[of L S K i M1 M2] backtrack(2,7,8,9) decomp*  
**by** (*fastforce simp add: lits-of-def*)  
**moreover** **then** **have** *undefined-lit M1 L*  
**by** (*simp add: defined-lit-map*)  
**moreover** **then** **have** *no-dup (trail T)*  
**using** *T decomp n-d* **by** (*auto simp: defined-lit-map M*)  
**ultimately** **show** *?case*  
**using** *T n-d unfolding M* **by** (*auto dest!: append-cons-eq-upt-length simp del: upt-simps*)  
**qed** *auto*

**lemma** *cdcl<sub>W</sub>-rf-bt:*

```

assumes
  cdclW-rf S S' and
  backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and
  get-all-levels-of-marked (trail S) = rev [1.. $\langle 1 + \text{length (get-all-levels-of-marked (trail S))} \rangle$ ]
shows backtrack-lvl S' = length (get-all-levels-of-marked (trail S'))
using assms by (induct rule: cdclW-rf.induct) auto

lemma cdclW-bt:
assumes
  cdclW S S' and
  backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and
  get-all-levels-of-marked (trail S)
  = rev ([1.. $\langle 1 + \text{length (get-all-levels-of-marked (trail S))} \rangle$ ]) and
  no-dup (trail S)
shows backtrack-lvl S' = length (get-all-levels-of-marked (trail S'))
using assms by (induct rule: cdclW.induct) (auto simp add: cdclW-o-bt cdclW-rf-bt)

lemma cdclW-bt-level':
assumes
  cdclW S S' and
  backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and
  get-all-levels-of-marked (trail S)
  = rev ([1.. $\langle 1 + \text{length (get-all-levels-of-marked (trail S))} \rangle$ ]) and
  n-d: no-dup (trail S)
shows get-all-levels-of-marked (trail S')
  = rev ([1.. $\langle 1 + \text{length (get-all-levels-of-marked (trail S'))} \rangle$ ])
using assms
proof (induct rule: cdclW-all-induct)
case (decide L T) note undef = this(2) and T = this(4)
let ?k = backtrack-lvl S
let ?M = trail S
let ?M' = Marked L (?k + 1) # trail S
have H: get-all-levels-of-marked ?M = rev [Suc 0.. $\langle 1 + \text{length (get-all-levels-of-marked ?M)} \rangle$ ]
  using decide.prems by simp
have k: ?k = length (get-all-levels-of-marked ?M)
  using decide.prems by auto
have get-all-levels-of-marked ?M' = Suc ?k # get-all-levels-of-marked ?M by simp
then have get-all-levels-of-marked ?M' = Suc ?k #
  rev [Suc 0.. $\langle 1 + \text{length (get-all-levels-of-marked ?M)} \rangle$ ]
  using H by auto
moreover have  $\dots = \text{rev [Suc 0.. $\langle \text{Suc (1 + length (get-all-levels-of-marked ?M))} \rangle$ ]}$ 
  unfolding k by simp
finally show ?case using T undef by (auto simp add: defined-lit-map)
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and confli = this(2) and T = this(6)
and
  all-marked = this(8) and bt-lvl = this(7)
have atm-of L  $\notin$  ( $\lambda l. \text{atm-of (lit-of l)}$ ) ' set M1
  using backtrack-lit-skipped[of L S K i M1 M2] backtrack(2,7,8,9) decomp
  by (fastforce simp add: lits-of-def)
moreover then have undefined-lit M1 L
  by (simp add: defined-lit-map)
then have [simp]: trail T = Propagated L (D + {#L#}) # M1
  using T decomp n-d by auto
obtain c where M: trail S = c @ M2 @ Marked K (i + 1) # M1 using decomp by auto

```

```

have get-all-levels-of-marked (rev (trail S))
  = [Suc 0.. $2 + \text{length (get-all-levels-of-marked } c) + (\text{length (get-all-levels-of-marked } M2) + \text{length (get-all-levels-of-marked } M1))$ ]
  using all-marked bt-lvl unfolding M by (auto simp add: rev-swap[symmetric] simp del: upt-simps)
then show ?case
  using T by (auto simp add: rev-swap M dest!: append-cons-eq-upt(1) simp del: upt-simps)
qed auto

```

We write  $1 + \text{length (get-all-levels-of-marked (trail S))}$  instead of *backtrack-lvl S* to avoid non termination of rewriting.

**definition** *cdcl<sub>W</sub>-M-level-inv* ( $S :: 'st$ )  $\longleftrightarrow$   
*consistent-interp* (lits-of (trail S))  
 $\wedge$  *no-dup* (trail S)  
 $\wedge$  *backtrack-lvl S* =  $\text{length (get-all-levels-of-marked (trail S))}$   
 $\wedge$  *get-all-levels-of-marked* (trail S)  
 =  $\text{rev ([1.. $1 + \text{length (get-all-levels-of-marked (trail S))}$ ])}$

**lemma** *cdcl<sub>W</sub>-M-level-inv-decomp*:  
**assumes** *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *consistent-interp* (lits-of (trail S))  
**and** *no-dup* (trail S)  
**using** *assms* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *fastforce*+

**lemma** *cdcl<sub>W</sub>-consistent-inv*:  
**fixes**  $S S' :: 'st$   
**assumes**  
   *cdcl<sub>W</sub> S S'* **and**  
   *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *cdcl<sub>W</sub>-M-level-inv S'*  
**using** *assms* *cdcl<sub>W</sub>-consistent-inv-2* *cdcl<sub>W</sub>-distinctinv-1* *cdcl<sub>W</sub>-bt* *cdcl<sub>W</sub>-bt-level'*  
**unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *meson*+

**lemma** *rtrancpl-cdcl<sub>W</sub>-consistent-inv*:  
**assumes** *cdcl<sub>W</sub>\*\* S S'*  
**and** *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *cdcl<sub>W</sub>-M-level-inv S'*  
**using** *assms* **by** (induct rule: *rtrancpl-induct*)  
 (auto intro: *cdcl<sub>W</sub>-consistent-inv*)

**lemma** *trancpl-cdcl<sub>W</sub>-consistent-inv*:  
**assumes** *cdcl<sub>W</sub><sup>++</sup> S S'*  
**and** *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *cdcl<sub>W</sub>-M-level-inv S'*  
**using** *assms* **by** (induct rule: *trancpl-induct*)  
 (auto intro: *cdcl<sub>W</sub>-consistent-inv*)

**lemma** *cdcl<sub>W</sub>-M-level-inv-S0-cdcl<sub>W</sub>[simp]*:  
*cdcl<sub>W</sub>-M-level-inv (init-state N)*  
**unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*

**lemma** *cdcl<sub>W</sub>-M-level-inv-get-level-le-backtrack-lvl*:  
**assumes** *inv: cdcl<sub>W</sub>-M-level-inv S*  
**shows** *get-level L (trail S)  $\leq$  backtrack-lvl S*

**proof** –  
**have** *get-all-levels-of-marked (trail S) = rev [1.. $1 + \text{backtrack-lvl S}$ ]*

```

    using inv unfolding cdclW-M-level-inv-def by auto
  then show ?thesis
    using get-rev-level-less-max-get-all-levels-of-marked[of L 0 rev (trail S)]
    by (auto simp: Max-n-upt)
qed

lemma backtrack-ex-decomp:
  assumes M-l: cdclW-M-level-inv S
  and i-S: i < backtrack-lvl S
  shows  $\exists K M1 M2. (\text{Marked } K (i + 1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
proof -
  let ?M = trail S
  have
    g: get-all-levels-of-marked (trail S) = rev [Suc 0.. $\text{Suc } (\text{backtrack-lvl } S)$ ]
    using M-l unfolding cdclW-M-level-inv-def by simp-all
  then have i+1  $\in \text{set } (\text{get-all-levels-of-marked } (\text{trail } S))$ 
    using i-S by auto

  then obtain c K c' where tr-S: trail S = c @ Marked K (i + 1) # c'
    using in-get-all-levels-of-marked-iff-decomp[of i+1 trail S] by auto

  obtain M1 M2 where (Marked K (i + 1) # M1, M2)  $\in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
    unfolding tr-S apply (induct c rule: marked-lit-list-induct)
    apply auto[2]
    apply (case-tac hd (get-all-marked-decomposition (xs @ Marked K (Suc i) # c')))
    apply (case-tac get-all-marked-decomposition (xs @ Marked K (Suc i) # c'))
    by auto
  then show ?thesis by blast
qed

```

### 17.4.2 Better-Suited Induction Principle

Ew generalise the induction principle defined previously: the induction case for *backtrack* now includes the assumption that *undefined-lit M1 L*. This helps the simplifier and thus the automation.

**lemma** *backtrack-induction-lev*[consumes 1, case-names *M-devel-inv backtrack*]:

```

  assumes
    bt: backtrack S T and
    inv: cdclW-M-level-inv S and
    backtrackH:  $\bigwedge K i M1 M2 L D T.
      (\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))
      \implies \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S
      \implies \text{conflicting } S = \text{C-Clause } (D + \{\#L\# \})
      \implies \text{get-level } L (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)
      \implies \text{get-maximum-level } D (\text{trail } S) \equiv i
      \implies \text{undefined-lit } M1 L
      \implies T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))
        (\text{reduce-trail-to } M1
          (\text{add-learned-cls } (D + \{\#L\# \})
            (\text{update-backtrack-lvl } i
              (\text{update-conflicting } \text{C-True } S))))
      \implies P S T
  shows P S T
proof -
  obtain K i M1 M2 L D where$ 
```



*decomp*: (*Marked* *K* (*Suc* *i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail* *S*)) **and**  
*L*: *get-level* *L* (*trail* *S*) = *backtrack-lvl* *S* **and**  
*confl*: *conflicting* *S* = *C-Clause* (*D* + {#*L*#}) **and**  
*lev-L*: *get-level* *L* (*trail* *S*) = *get-maximum-level* (*D*+{#*L*#}) (*trail* *S*) **and**  
*lev-D*: *get-maximum-level* *D* (*trail* *S*) ≡ *i* **and**  
*T*: *T* ∼ *cons-trail* (*Propagated* *L* (*D*+{#*L*#}))  
    (*reduce-trail-to* *M1*  
    (*add-learned-cl*s (*D* + {#*L*#})  
    (*update-backtrack-lvl* *i*  
    (*update-conflicting* *C-True* *S*))))  
**using** *bt* **by** (*elim backtrackE*) *metis*  
  
**have** *atm-of* *L* ∉ (*λl. atm-of* (*lit-of* *l*)) ‘ *set* *M1*  
**using** *backtrack-lit-skipped*[*of* *L* *S* *K* *i* *M1* *M2*] *L decomp bt confl lev-L lev-D inv*  
**unfolding** *cdcl<sub>W</sub>-M-level-inv-def*  
**by** (*fastforce simp add: lits-of-def*)  
**then have** *undefined-lit* *M1* *L*  
**by** (*auto simp: defined-lit-map*)  
**then show** ?*thesis*  
**using** *backtrackH*[*OF decomp L confl lev-L lev-D - T*] **by** *simp*  
**qed**

**lemmas** *backtrack-induction-lev2* = *backtrack-induction-lev*[*consumes 2, case-names backtrack*]

**lemma** *cdcl<sub>W</sub>-all-induct-lev-full*:

**fixes** *S* :: 'st

**assumes**

*cdcl<sub>W</sub>*: *cdcl<sub>W</sub>* *S* *S'* **and**

*inv*[*simp*]: *cdcl<sub>W</sub>-M-level-inv* *S* **and**

*propagateH*:  $\bigwedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \implies \text{trail } S \models_{as} C \text{Not } C$

$\implies \text{undefined-lit } (\text{trail } S) L \implies \text{conflicting } S = C\text{-True}$

$\implies T \sim \text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$  **and**

*conflictH*:  $\bigwedge D T. D \in \# \text{ clauses } S \implies \text{conflicting } S = C\text{-True} \implies \text{trail } S \models_{as} C \text{Not } D$

$\implies T \sim \text{update-conflicting } (C\text{-Clause } D) S$

$\implies P S T$  **and**

*forgetH*:  $\bigwedge C T. \neg \text{trail } S \models_{asm} \text{clauses } S$

$\implies C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$

$\implies C \notin \# \text{ init-clss } S$

$\implies C \in \# \text{ learned-clss } S$

$\implies \text{conflicting } S = C\text{-True}$

$\implies T \sim \text{remove-cl} s C S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$  **and**

*restartH*:  $\bigwedge T. \neg \text{trail } S \models_{asm} \text{clauses } S$

$\implies \text{conflicting } S = C\text{-True}$

$\implies T \sim \text{restart-state } S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$  **and**

*decideH*:  $\bigwedge L T. \text{conflicting } S = C\text{-True} \implies \text{undefined-lit } (\text{trail } S) L$

$\implies \text{atm-of } L \in \text{atms-of-mu } (\text{init-clss } S)$

$\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$  **and**

```

skipH:  $\bigwedge L C' M D T. \text{trail } S = \text{Propagated } L C' \# M$ 
 $\implies \text{conflicting } S = C\text{-Clause } D \implies -L \notin \# D \implies D \neq \{\#\}$ 
 $\implies T \sim \text{tl-trail } S$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$  and
resolveH:  $\bigwedge L C M D T.$ 
 $\text{trail } S = \text{Propagated } L ( (C + \{\#L\# \}) \# M$ 
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ 
 $\implies \text{get-maximum-level } D (\text{Propagated } L ( (C + \{\#L\# \}) \# M) = \text{backtrack-lvl } S$ 
 $\implies T \sim (\text{update-conflicting } (C\text{-Clause } (D \# \cup C)) (\text{tl-trail } S))$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$  and
backtrackH:  $\bigwedge K i M1 M2 L D T.$ 
 $(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
 $\implies \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S$ 
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ 
 $\implies \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S) = \text{get-level } L (\text{trail } S)$ 
 $\implies \text{get-maximum-level } D (\text{trail } S) \equiv i$ 
 $\implies \text{undefined-lit } M1 L$ 
 $\implies T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$ 
 $\quad (\text{reduce-trail-to } M1$ 
 $\quad \quad (\text{add-learned-cls } (D + \{\#L\# \})$ 
 $\quad \quad \quad (\text{update-backtrack-lvl } i$ 
 $\quad \quad \quad \quad (\text{update-conflicting } C\text{-True } S))))$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$ 
shows  $P S S'$ 
using  $\text{cdcl}_W$ 
proof (induct  $S'$  rule:  $\text{cdcl}_W\text{-all-rules-induct}$ )
case (propagate  $S'$ )
then show ?case by (elim propagateE) (frule propagateH; simp)
next
case (conflict  $S'$ )
then show ?case by (elim conflictE) (frule conflictH; simp)
next
case (restart  $S'$ )
then show ?case by (elim restartE) (frule restartH; simp)
next
case (decide  $T$ )
then show ?case by (elim decideE) (frule decideH; simp)
next
case (backtrack  $S'$ )
then show ?case
apply (induction rule: backtrack-induction-lev)
apply (rule inv)
by (rule backtrackH;
 $\text{fastforce simp del: state-simp simp add: state-eq-def dest!: HOL.meta-eq-to-obj-eq}$ )
next
case (forget  $S'$ )
then show ?case using forgetH by auto
next
case (skip  $S'$ )
then show ?case using skipH by auto
next
case (resolve  $S'$ )

```

```

then show ?case by (elim resolveE) (frule resolveH; simp)
qed

lemmas cdclW-all-induct-lev2 = cdclW-all-induct-lev-full[consumes 2, case-names propagate conflict
forget restart decide skip resolve backtrack]

lemmas cdclW-all-induct-lev = cdclW-all-induct-lev-full[consumes 1, case-names lev-inv propagate
conflict forget restart decide skip resolve backtrack]

thm cdclW-o-induct
lemma cdclW-o-induct-lev[consumes 1, case-names M-lev decide skip resolve backtrack]:
fixes S :: 'st
assumes
  cdclW: cdclW-o S T and
  inv[simp]: cdclW-M-level-inv S and
  decideH:  $\bigwedge L T. \text{conflicting } S = C\text{-True} \implies \text{undefined-lit } (\text{trail } S) L$ 
 $\implies \text{atm-of } L \in \text{atms-of-mu } (\text{init-clss } S)$ 
 $\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$  and
  skipH:  $\bigwedge L C' M D T. \text{trail } S = \text{Propagated } L C' \# M$ 
 $\implies \text{conflicting } S = C\text{-Clause } D \implies -L \notin \# D \implies D \neq \{\#\}$ 
 $\implies T \sim \text{tl-trail } S$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$  and
  resolveH:  $\bigwedge L C M D T.$ 
 $\text{trail } S = \text{Propagated } L ( (C + \{\#L\# \}) \# M$ 
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#-L\# \})$ 
 $\implies \text{get-maximum-level } D (\text{Propagated } L (C + \{\#L\# \}) \# M) = \text{backtrack-lvl } S$ 
 $\implies T \sim \text{update-conflicting } (C\text{-Clause } (D \# \cup C)) (\text{tl-trail } S)$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$  and
  backtrackH:  $\bigwedge K i M1 M2 L D T.$ 
 $(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
 $\implies \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S$ 
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ 
 $\implies \text{get-level } L (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$ 
 $\implies \text{get-maximum-level } D (\text{trail } S) \equiv i$ 
 $\implies \text{undefined-lit } M1 L$ 
 $\implies T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$ 
 $(\text{reduce-trail-to } M1$ 
 $(\text{add-learned-cls } (D + \{\#L\# \})$ 
 $(\text{update-backtrack-lvl } i$ 
 $(\text{update-conflicting } C\text{-True } S))))$ 
 $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
 $\implies P S T$ 
shows P S T
using cdclW
proof (induct S T rule: cdclW-o-all-rules-induct)
case (decide T)
then show ?case by (elim decideE) (frule decideH; simp)
next
case (backtrack S')
then show ?case
using inv apply (induction rule: backtrack-induction-lev2)

```

```

    by (rule backtrackH)
      (fastforce simp del: state-simp simp add: state-eq-def dest!: HOL.meta-eq-to-obj-eq)+
next
case (skip S')
then show ?case using skipH by auto
next
case (resolve S')
then show ?case by (elim resolveE) (frule resolveH; simp)
qed

lemmas cdclW-o-induct-lev2 = cdclW-o-induct-lev[consumes 2, case-names decide skip resolve
backtrack]

```

### 17.4.3 Compatibility with $op \sim$

```

lemma propagate-state-eq-compatible:
  assumes
    propagate S T and
    S  $\sim$  S' and
    T  $\sim$  T'
  shows propagate S' T'
  using assms apply (elim propagateE)
  apply (rule propagate-rule)
  by (auto simp: state-eq-def clauses-def simp del: state-simp)

```

```

lemma conflict-state-eq-compatible:
  assumes
    conflict S T and
    S  $\sim$  S' and
    T  $\sim$  T'
  shows conflict S' T'
  using assms apply (elim conflictE)
  apply (rule conflict-rule)
  by (auto simp: state-eq-def clauses-def simp del: state-simp)

```

```

lemma backtrack-state-eq-compatible:
  assumes
    backtrack S T and
    S  $\sim$  S' and
    T  $\sim$  T' and
    inv: cdclW-M-level-inv S
  shows backtrack S' T'
  using assms apply (induction rule: backtrack-induction-lev)
  using inv apply simp
  apply (rule backtrack-rule)
  apply auto[5]
  by (auto simp: state-eq-def clauses-def cdclW-M-level-inv-def simp del: state-simp)

```

```

lemma decide-state-eq-compatible:
  assumes
    decide S T and
    S  $\sim$  S' and
    T  $\sim$  T'
  shows decide S' T'
  using assms apply (elim decideE)
  apply (rule decide-rule)

```

**by** (auto simp: state-eq-def clauses-def simp del: state-simp)

**lemma** skip-state-eq-compatible:

**assumes**  
 skip  $S$   $T$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$   
**shows** skip  $S'$   $T'$   
**using** assms **apply** (elim skipE)  
**apply** (rule skip-rule)  
**by** (auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of - # - trail -]  
 simp del: state-simp dest: arg-cong[of - # trail - trail - tl])

**lemma** resolve-state-eq-compatible:

**assumes**  
 resolve  $S$   $T$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$   
**shows** resolve  $S'$   $T'$   
**using** assms **apply** (elim resolveE)  
**apply** (rule resolve-rule)  
**by** (auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of - # - trail -]  
 simp del: state-simp dest: arg-cong[of - # trail - trail - tl])

**lemma** forget-state-eq-compatible:

**assumes**  
 forget  $S$   $T$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$   
**shows** forget  $S'$   $T'$   
**using** assms **apply** (elim forgetE)  
**apply** (rule forget-rule)  
**by** (auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of {#-#} + - -]  
 simp del: state-simp dest: arg-cong[of - # trail - trail - tl])

**lemma** cdcl<sub>W</sub>-state-eq-compatible:

**assumes**  
 cdcl<sub>W</sub>  $S$   $T$  **and**  $\neg$ restart  $S$   $T$  **and**  
 $S \sim S'$  **and**  
 $T \sim T'$  **and**  
 inv: cdcl<sub>W</sub>-M-level-inv  $S$   
**shows** cdcl<sub>W</sub>  $S'$   $T'$   
**using** assms **by** (meson assms backtrack-state-eq-compatible bj cdcl<sub>W</sub>.simps cdcl<sub>W</sub>-bj.simps  
 cdcl<sub>W</sub>-o-rule-cases cdcl<sub>W</sub>-rf.cases cdcl<sub>W</sub>-rf.restart conflict-state-eq-compatible decide  
 decide-state-eq-compatible forget forget-state-eq-compatible  
 propagate-state-eq-compatible resolve-state-eq-compatible  
 skip-state-eq-compatible)

#### 17.4.4 Conservation of some Properties

**lemma** level-of-marked-ge-1:

**assumes**  
 cdcl<sub>W</sub>  $S$   $S'$  **and**  
 inv: cdcl<sub>W</sub>-M-level-inv  $S$  **and**  
 $\forall L\ l. \text{Marked } L\ l \in \text{set } (\text{trail } S) \longrightarrow l > 0$   
**shows**  $\forall L\ l. \text{Marked } L\ l \in \text{set } (\text{trail } S') \longrightarrow l > 0$

**using** *assms* **apply** (*induct rule: cdcl<sub>W</sub>-all-induct-lev2*)  
**by** (*auto dest: union-in-get-all-marked-decomposition-is-subset simp: cdcl<sub>W</sub>-M-level-inv-decomp*)

**lemma** *cdcl<sub>W</sub>-o-no-more-init-clss:*

**assumes**

*cdcl<sub>W</sub>-o S S' and*

*inv: cdcl<sub>W</sub>-M-level-inv S*

**shows** *init-clss S = init-clss S'*

**using** *assms* **by** (*induct rule: cdcl<sub>W</sub>-o-induct-lev2*) (*auto simp: cdcl<sub>W</sub>-M-level-inv-decomp*)

**lemma** *trancpl-cdcl<sub>W</sub>-o-no-more-init-clss:*

**assumes**

*cdcl<sub>W</sub>-o<sup>++</sup> S S' and*

*inv: cdcl<sub>W</sub>-M-level-inv S*

**shows** *init-clss S = init-clss S'*

**using** *assms* **apply** (*induct rule: trancpl.induct*)

**by** (*auto dest: cdcl<sub>W</sub>-o-no-more-init-clss*

*dest!: trancpl-cdcl<sub>W</sub>-consistent-inv dest: trancpl-mono-explicit[of cdcl<sub>W</sub>-o - - cdcl<sub>W</sub>]*

*simp: other*)

**lemma** *rtrancpl-cdcl<sub>W</sub>-o-no-more-init-clss:*

**assumes**

*cdcl<sub>W</sub>-o<sup>\*\*</sup> S S' and*

*inv: cdcl<sub>W</sub>-M-level-inv S*

**shows** *init-clss S = init-clss S'*

**using** *assms* **unfolding** *rtrancpl-unfold* **by** (*auto intro: trancpl-cdcl<sub>W</sub>-o-no-more-init-clss*)

**lemma** *cdcl<sub>W</sub>-init-clss:*

*cdcl<sub>W</sub> S T  $\implies$  cdcl<sub>W</sub>-M-level-inv S  $\implies$  init-clss S = init-clss T*

**by** (*induct rule: cdcl<sub>W</sub>-all-induct-lev2*) (*auto simp: cdcl<sub>W</sub>-M-level-inv-def*)

**lemma** *rtrancpl-cdcl<sub>W</sub>-init-clss:*

*cdcl<sub>W</sub><sup>\*\*</sup> S T  $\implies$  cdcl<sub>W</sub>-M-level-inv S  $\implies$  init-clss S = init-clss T*

**by** (*induct rule: rtrancpl-induct*) (*auto dest: cdcl<sub>W</sub>-init-clss rtrancpl-cdcl<sub>W</sub>-consistent-inv*)

**lemma** *trancpl-cdcl<sub>W</sub>-init-clss:*

*cdcl<sub>W</sub><sup>++</sup> S T  $\implies$  cdcl<sub>W</sub>-M-level-inv S  $\implies$  init-clss S = init-clss T*

**using** *rtrancpl-cdcl<sub>W</sub>-init-clss[of S T]* **unfolding** *rtrancpl-unfold* **by** *auto*

### 17.4.5 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. A more precise version would be to show that either these marked are learned or are in the set of clauses

**definition** *cdcl<sub>W</sub>-learned-clause (S:: 'st)  $\longleftrightarrow$*

*(init-clss S  $\models_{psm}$  learned-clss S*

$\wedge (\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{init-clss } S \models_{pm} T)$

$\wedge \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \subseteq \text{set-mset } (\text{clauses } S))$

```

lemma cdclW-learned-clause-S0-cdclW[simp]:
  cdclW-learned-clause (init-state N)
  unfolding cdclW-learned-clause-def by auto

lemma cdclW-learned-clss:
  assumes
    cdclW S S' and
    learned: cdclW-learned-clause S and
    lev-inv: cdclW-M-level-inv S
  shows cdclW-learned-clause S'
  using assms(1) lev-inv learned
proof (induct rule: cdclW-all-induct-lev2)
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and confl = this(3) and undef = this(6)
  and T = this(7)
  show ?case
    using decomp confl learned undef T lev-inv unfolding cdclW-learned-clause-def
    by (auto dest!: get-all-marked-decomposition-exists-prepend
      simp: clauses-def cdclW-M-level-inv-decomp dest: true-clss-clss-left-right)
  next
  case (resolve L C M D) note trail = this(1) and confl = this(2) and lvl = this(3) and
    T = this(4)
  moreover
    have init-clss S ⊨psm learned-clss S
    using learned trail unfolding cdclW-learned-clause-def clauses-def by auto
    then have init-clss S ⊨pm C + {#L#}
    using trail learned unfolding cdclW-learned-clause-def clauses-def
    by (auto dest: true-clss-clss-in-imp-true-clss-clss)
  ultimately show ?case
    using learned
    by (auto dest: mk-disjoint-insert true-clss-clss-left-right
      simp add: cdclW-learned-clause-def clauses-def
      intro: true-clss-clss-union-mset-true-clss-clss-or-not-true-clss-clss-or)
  next
  case (restart T)
  then show ?case
    using learned-clss-restart-state[of T]
    by (auto dest!: get-all-marked-decomposition-exists-prepend
      simp: clauses-def state-eq-def cdclW-learned-clause-def
      simp del: state-simp
      dest: true-clss-clssm-subsetE)
  next
  case propagate
  then show ?case using learned by (auto simp: cdclW-learned-clause-def clauses-def)
  next
  case conflict
  then show ?case using learned
    by (auto simp: cdclW-learned-clause-def clauses-def true-clss-clss-in-imp-true-clss-clss)
  next
  case forget
  then show ?case
    using learned by (auto simp: cdclW-learned-clause-def clauses-def split: split-if-asm)
qed (auto simp: cdclW-learned-clause-def clauses-def)

lemma rtrancpl-cdclW-learned-clss:

```

**assumes**  
 $cdcl_W^{**} S S'$  **and**  
 $cdcl_W$ - $M$ -level-inv  $S$   
 $cdcl_W$ -learned-clause  $S$   
**shows**  $cdcl_W$ -learned-clause  $S'$   
**using** *assms* **by** *induction* (*auto dest: cdcl\_W-learned-clss intro: rtranclp-cdcl\_W-consistent-inv*)

#### 17.4.6 No alien atom in the state

This invariant means that all the literals are in the set of clauses.

**definition**  $no\text{-}strange\text{-}atm S' \longleftrightarrow$  (  
 $(\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mu } (init\text{-}clss S'))$   
 $\wedge (\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (trail S')$   
 $\longrightarrow \text{atms-of } (mark) \subseteq \text{atms-of-mu } (init\text{-}clss S'))$   
 $\wedge \text{atms-of-mu } (learned\text{-}clss S') \subseteq \text{atms-of-mu } (init\text{-}clss S')$   
 $\wedge \text{atm-of } ' (lits\text{-of } (trail S')) \subseteq \text{atms-of-mu } (init\text{-}clss S'))$

**lemma**  $no\text{-}strange\text{-}atm\text{-}decomp$ :

**assumes**  $no\text{-}strange\text{-}atm S$   
**shows**  $\text{conflicting } S = C\text{-Clause } T \implies \text{atms-of } T \subseteq \text{atms-of-mu } (init\text{-}clss S)$   
**and**  $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (trail S)$   
 $\longrightarrow \text{atms-of } (mark) \subseteq \text{atms-of-mu } (init\text{-}clss S))$   
**and**  $\text{atms-of-mu } (learned\text{-}clss S) \subseteq \text{atms-of-mu } (init\text{-}clss S)$   
**and**  $\text{atm-of } ' (lits\text{-of } (trail S)) \subseteq \text{atms-of-mu } (init\text{-}clss S)$   
**using** *assms* **unfolding**  $no\text{-}strange\text{-}atm\text{-}def$  **by** *blast+*

**lemma**  $no\text{-}strange\text{-}atm\text{-}S0$  [*simp*]:  $no\text{-}strange\text{-}atm (init\text{-}state N)$   
**unfolding**  $no\text{-}strange\text{-}atm\text{-}def$  **by** *auto*

**lemma**  $cdcl_W\text{-}no\text{-}strange\text{-}atm\text{-}explicit$ :

**assumes**  
 $cdcl_W S S'$  **and**  
 $lev$ :  $cdcl_W$ - $M$ -level-inv  $S$  **and**  
 $conf$ :  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mu } (init\text{-}clss S)$  **and**  
 $marked$ :  $\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (trail S)$   
 $\longrightarrow \text{atms-of } mark \subseteq \text{atms-of-mu } (init\text{-}clss S)$  **and**  
 $learned$ :  $\text{atms-of-mu } (learned\text{-}clss S) \subseteq \text{atms-of-mu } (init\text{-}clss S)$  **and**  
 $trail$ :  $\text{atm-of } ' (lits\text{-of } (trail S)) \subseteq \text{atms-of-mu } (init\text{-}clss S)$   
**shows**  $(\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mu } (init\text{-}clss S')) \wedge$   
 $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (trail S')$   
 $\longrightarrow \text{atms-of } (mark) \subseteq \text{atms-of-mu } (init\text{-}clss S')) \wedge$   
 $\text{atms-of-mu } (learned\text{-}clss S') \subseteq \text{atms-of-mu } (init\text{-}clss S') \wedge$   
 $\text{atm-of } ' (lits\text{-of } (trail S')) \subseteq \text{atms-of-mu } (init\text{-}clss S') \text{ (is } ?C S' \wedge ?M S' \wedge ?U S' \wedge ?V S')$   
**using** *assms*(1,2)

**proof** (*induct rule: cdcl\_W-all-induct-lev2*)

**case** ( $\text{propagate } C L T$ ) **note**  $C\text{-}L = \text{this}(1)$  **and**  $undef = \text{this}(3)$  **and**  $confl = \text{this}(4)$  **and**  $T = \text{this}(5)$   
**have**  $?C (cons\text{-}trail (\text{Propagated } L (C + \{\#L\# \})) S)$  **using** *confl undef* **by** *auto*

**moreover**

**have**  $\text{atms-of } (C + \{\#L\# \}) \subseteq \text{atms-of-mu } (init\text{-}clss S)$   
**by** (*metis* (*no-types*) *atms-of-atms-of-m-mono* *atms-of-m-union clauses-def mem-set-mset-iff*  
 $C\text{-}L \text{ learned set-mset-union sup.orderE}$ )  
**then have**  $?M (cons\text{-}trail (\text{Propagated } L (C + \{\#L\# \})) S)$  **using** *undef*  
**by** (*simp add: marked*)

**moreover have**  $?U (cons\text{-}trail (\text{Propagated } L (C + \{\#L\# \})) S)$   
**using** *learned undef* **by** *auto*



```

moreover have ?V (cons-trail (Propagated L (C + {#L#})) S)
  using C-L learned trail undef unfolding clauses-def
  by (auto simp: in-plus-implies-atm-of-on-atms-of-m)
ultimately show ?case using T by auto
next
  case (decide L)
  then show ?case using learned marked conf trail unfolding clauses-def by auto
next
  case (skip L C M D)
  then show ?case using learned marked conf trail by auto
next
  case (conflict D T) note T = this(4)
  have D: atm-of ' set-mset D  $\subseteq \bigcup (\text{atms-of ' (set-mset (clauses S))})$ 
    using (D  $\in \#$  clauses S) by (auto simp add: atms-of-def atms-of-m-def)
  moreover {
    fix xa :: 'v literal
    assume a1: atm-of ' set-mset D  $\subseteq (\bigcup x \in \text{set-mset (init-clss S). atms-of x})$ 
       $\cup (\bigcup x \in \text{set-mset (learned-clss S). atms-of x})$ 
    assume a2:  $(\bigcup x \in \text{set-mset (learned-clss S). atms-of x}) \subseteq (\bigcup x \in \text{set-mset (init-clss S). atms-of x})$ 
    assume xa  $\in \#$  D
    then have atm-of xa  $\in \text{UNION (set-mset (init-clss S)) atms-of}$ 
      using a2 a1 by (metis (no-types) Un-iff atm-of-lit-in-atms-of atms-of-def subset-Un-eq)
    then have  $\exists m \in \text{set-mset (init-clss S). atm-of xa} \in \text{atms-of m}$ 
      by blast
  } note H = this
  ultimately show ?case using conflict.premis T learned marked conf trail
    unfolding atms-of-def atms-of-m-def clauses-def
    by (auto simp add: H )
next
  case (restart T)
  then show ?case using learned marked conf trail by auto
next
  case (forget C T) note C = this(3) and C-le = this(4) and confl = this(5) and
    T = this(6)
  have H:  $\bigwedge L \text{ mark. Propagated L mark} \in \text{set (trail S)} \implies \text{atms-of mark} \subseteq \text{atms-of-mu (init-clss S)}$ 
    using marked by simp
  show ?case unfolding clauses-def apply standard
    using conf T trail C unfolding clauses-def apply (auto dest!: H)[]
    apply standard
    using T trail C apply (auto dest!: H)[]
    apply standard
    using T learned C C-le atms-of-m-remove-subset[of set-mset (learned-clss S)] apply (auto)[]
    using T trail C apply (auto simp: clauses-def lits-of-def)[]
  done
next
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and confl = this(3) and undef = this(6)
    and T = this(7)
  have ?C T
    using conf T decomp undef lev by (auto simp: cdclW-M-level-inv-decomp)
  moreover have set M1  $\subseteq \text{set (trail S)}$ 
    using backtrack.hyps(1) by auto
  then have M: ?M T
    using marked conf undef confl T decomp lev
    by (auto simp: image-subset-iff clauses-def cdclW-M-level-inv-decomp)
  moreover have ?U T

```

**using** *learned decomp conf confl T undef lev* **unfolding** *clauses-def*  
**by** (*auto simp: cdcl<sub>W</sub>-M-level-inv-decomp*)  
**moreover have**  $?V \ T$   
**using** *M conf confl trail T undef decomp lev* **by** (*force simp: cdcl<sub>W</sub>-M-level-inv-decomp*)  
**ultimately show**  $?case$  **by** *blast*  
**next**  
**case** (*resolve L C M D T*) **note** *trail-S = this(1)* **and** *confl = this(2)* **and** *T = this(4)*  
**let**  $?T = \text{update-conflicting } (C\text{-Clause } (\text{remdups-mset } (D + C))) \ (tl\text{-trail } S)$   
**have**  $?C \ ?T$   
**using** *confl trail-S conf marked by simp*  
**moreover have**  $?M \ ?T$   
**using** *confl trail-S conf marked by auto*  
**moreover have**  $?U \ ?T$   
**using** *trail learned by auto*  
**moreover have**  $?V \ ?T$   
**using** *confl trail-S trail by auto*  
**ultimately show**  $?case$  **using** *T* **by** *auto*  
**qed**

**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'*  
**using** *cdcl<sub>W</sub>-no-strange-atm-explicit[OF assms(1)] assms(2,3)* **unfolding** *no-strange-atm-def* **by** *fast*

**lemma** *rtrancpl-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'*  
**using** *assms by induction (auto intro: cdcl<sub>W</sub>-no-strange-atm-inv rtrancpl-cdcl<sub>W</sub>-consistent-inv)*

#### 17.4.7 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 moreover.

**definition** *distinct-cdcl<sub>W</sub>-state (S::'st)*  
 $\longleftrightarrow ((\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{distinct-mset } T)$   
 $\wedge \text{distinct-mset-mset } (\text{learned-clss } S)$   
 $\wedge \text{distinct-mset-mset } (\text{init-clss } S)$   
 $\wedge (\forall L \text{ mark. } (\text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset } (\text{mark}))))$

**lemma** *distinct-cdcl<sub>W</sub>-state-decomp:*  
**assumes** *distinct-cdcl<sub>W</sub>-state (S::'st)*  
**shows**  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{distinct-mset } T$   
**and** *distinct-mset-mset (learned-clss S)*  
**and** *distinct-mset-mset (init-clss S)*  
**and**  $\forall L \text{ mark. } (\text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset } (\text{mark}))$   
**using** *assms unfolding distinct-cdcl<sub>W</sub>-state-def by blast+*

**lemma** *distinct-cdcl<sub>W</sub>-state-decomp-2:*  
**assumes** *distinct-cdcl<sub>W</sub>-state (S::'st)*  
**shows** *conflicting S = C-Clause T  $\implies$  distinct-mset T*  
**using** *assms unfolding distinct-cdcl<sub>W</sub>-state-def by auto*

**lemma** *distinct-cdcl<sub>W</sub>-state-S0-cdcl<sub>W</sub>[simp]:*  
 $\text{distinct-mset-mset } N \implies \text{distinct-cdcl<sub>W</sub>-state } (\text{init-state } N)$   
**unfolding** *distinct-cdcl<sub>W</sub>-state-def by auto*

```

lemma distinct-cdclW-state-inv:
  assumes
    cdclW S S' and
    cdclW-M-level-inv S and
    distinct-cdclW-state S
  shows distinct-cdclW-state S'
  using assms
proof (induct rule: cdclW-all-induct-lev2)
  case (backtrack K i M1 M2 L D)
  then show ?case
    unfolding distinct-cdclW-state-def
    by (fastforce dest: get-all-marked-decomposition-incl simp: cdclW-M-level-inv-decomp)
next
  case restart
  then show ?case unfolding distinct-cdclW-state-def distinct-mset-set-def clauses-def
  using learned-clss-restart-state[of S] by auto
next
  case resolve
  then show ?case
    by (auto simp add: distinct-cdclW-state-def distinct-mset-set-def clauses-def
      distinct-mset-single-add
      intro!: distinct-mset-union-mset)
qed (auto simp add: distinct-cdclW-state-def distinct-mset-set-def clauses-def)

lemma rtanclp-distinct-cdclW-state-inv:
  assumes
    cdclW** S S' and
    cdclW-M-level-inv S and
    distinct-cdclW-state S
  shows distinct-cdclW-state S'
  using assms apply (induct rule: rtanclp-induct)
  using distinct-cdclW-state-inv rtanclp-cdclW-consistent-inv by blast+

```

#### 17.4.8 Conflicts and co

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-mark-is-a-conflict S*  $\equiv$

$\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark} \ \# \ b = (\text{trail } S)$   
 $\longrightarrow (b \models_{as} CNot \ ( \text{mark} - \{ \#L\} )) \wedge L \in \# \ \text{mark}$

**definition** *cdcl<sub>W</sub>-conflicting S*  $\equiv$

$(\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot \ T)$   
 $\wedge \text{every-mark-is-a-conflict } S$

**lemma** *backtrack-atms-of-D-in-M1*:

**fixes** *M1* :: (*'v*, *nat*, *'v clause*) *marked-lits*

**assumes**

*inv: cdcl<sub>W</sub>-M-level-inv S and*

*undef: undefined-lit M1 L and*

*i: get-maximum-level D (trail S) = i and*

*decomp: (Marked K (Suc i)  $\#$  M1, M2)*

$\in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$  **and**

*S-lvl*:  $\text{backtrack-lvl } S = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S) \text{ and}$   
*S-conflict*:  $\text{conflicting } S = \text{C-Clause } (D + \{\#L\# \}) \text{ and}$   
*undef*:  $\text{undefined-lit } M1 \text{ L and}$   
*T*:  $T \sim (\text{cons-trail } (\text{Propagated } L (D + \{\#L\# \})))$   
 $(\text{reduce-trail-to } M1$   
 $(\text{add-learned-cls } (D + \{\#L\# \}))$   
 $(\text{update-backtrack-lvl } i$   
 $(\text{update-conflicting } \text{C-True } S)))) \text{ and}$   
*conf*:  $\forall T. \text{conflicting } S = \text{C-Clause } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$   
**shows**  $\text{atms-of } D \subseteq \text{atm-of ' lits-of } (\text{tl } (\text{trail } T))$   
**proof** (rule *ccontr*)  
**let**  $?k = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$   
**have**  $\text{trail } S \models_{\text{as}} \text{CNot } D$  **using** *conf* *S-conf* **by** *auto*  
**then have**  $\text{vars-of-} D: \text{atms-of } D \subseteq \text{atm-of ' lits-of } (\text{trail } S)$  **unfolding** *atms-of-def*  
**by** (*meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined*)  
  
**obtain**  $M0$  **where**  $M: \text{trail } S = M0 @ M2 @ \text{Marked } K (\text{Suc } i) \# M1$   
**using** *decomp* **by** *auto*  
  
**have**  $\text{max}: \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$   
 $= \text{length } (\text{get-all-levels-of-marked } (M0 @ M2 @ \text{Marked } K (\text{Suc } i) \# M1))$   
**using** *inv* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def S-lvl M* **by** *simp*  
**assume**  $a: \neg ?thesis$   
**then obtain**  $L'$  **where**  
 $L': L' \in \text{atms-of } D$  **and**  
 $L'\text{-notin-}M1: L' \notin \text{atm-of ' lits-of } M1$   
**using** *T undef decomp inv* **by** (*auto simp: cdcl<sub>W</sub>-M-level-inv-decomp*)  
**then have**  $L'\text{-in}: L' \in \text{atm-of ' lits-of } (M0 @ M2 @ \text{Marked } K (i + 1) \# [])$   
**using** *vars-of-D* **unfolding** *M* **by** *force*  
**then obtain**  $L''$  **where**  
 $L'' \in \# D$  **and**  
 $L'': L' = \text{atm-of } L''$   
**using**  $L' L'\text{-notin-}M1$  **unfolding** *atms-of-def* **by** *auto*  
**have**  $\text{get-level } L'' (\text{trail } S) = \text{get-rev-level } L'' (\text{Suc } i) (\text{Marked } K (\text{Suc } i) \# \text{rev } M2 @ \text{rev } M0)$   
**using**  $L'\text{-notin-}M1 L'' M$  **by** (*auto simp del: get-rev-level.simps*)  
**have**  $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [1..<1+?k]$   
**using** *inv S-lvl* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**then have**  $\text{get-all-levels-of-marked } (M0 @ M2)$   
 $= \text{rev } [\text{Suc } (\text{Suc } i)..<\text{Suc } (\text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S))]$   
**unfolding** *M* **by** (*auto simp: rev-swap[symmetric] dest!: append-cons-eq-upt-length-i-end*)  
  
**then have**  $M: \text{get-all-levels-of-marked } M0 @ \text{get-all-levels-of-marked } M2$   
 $= \text{rev } [\text{Suc } (\text{Suc } i)..<\text{Suc } (\text{length } (\text{get-all-levels-of-marked } (M0 @ M2 @ \text{Marked } K (\text{Suc } i) \# M1)))]$   
**unfolding** *max* **unfolding** *M* **by** *simp*  
  
**have**  $\text{get-rev-level } L'' (\text{Suc } i) (\text{Marked } K (\text{Suc } i) \# \text{rev } (M0 @ M2))$   
 $\geq \text{Min } (\text{set } ((\text{Suc } i) \# \text{get-all-levels-of-marked } (\text{Marked } K (\text{Suc } i) \# \text{rev } (M0 @ M2))))$   
**using** *get-rev-level-ge-min-get-all-levels-of-marked[of L''*  
 $\text{rev } (M0 @ M2 @ [\text{Marked } K (\text{Suc } i)]] \text{ Suc } i] L'\text{-in}$   
**unfolding**  $L''$  **by** (*fastforce simp add: lits-of-def*)  
**also have**  $\text{Min } (\text{set } ((\text{Suc } i) \# \text{get-all-levels-of-marked } (\text{Marked } K (\text{Suc } i) \# \text{rev } (M0 @ M2))))$   
 $= \text{Min } (\text{set } ((\text{Suc } i) \# \text{get-all-levels-of-marked } (\text{rev } (M0 @ M2))))$  **by** *auto*  
**also have**  $\dots = \text{Min } (\text{set } ((\text{Suc } i) \# \text{get-all-levels-of-marked } M0 @ \text{get-all-levels-of-marked } M2))$   
**by** (*simp add: Un-commute*)  
**also have**  $\dots = \text{Min } (\text{set } ((\text{Suc } i) \# [\text{Suc } (\text{Suc } i)..<2 + \text{length } (\text{get-all-levels-of-marked } M0)]$

```

+ (length (get-all-levels-of-marked M2) + length (get-all-levels-of-marked M1))))
unfolding M by (auto simp add: Un-commute)
also have ... = Suc i by (auto intro: Min-eqI)
finally have get-rev-level L'' (Suc i) (Marked K (Suc i) # rev (M0 @ M2)) ≥ Suc i .
then have get-level L'' (trail S) ≥ i + 1
  using ⟨get-level L'' (trail S) = get-rev-level L'' (Suc i) (Marked K (Suc i) # rev M2 @ rev M0)⟩
  by simp
then have get-maximum-level D (trail S) ≥ i + 1
  using get-maximum-level-ge-get-level[OF ⟨L'' ∈ # D⟩, of trail S] by auto
then show False using i by auto
qed

```

**lemma** *distinct-atms-of-incl-not-in-other:*

```

assumes a1: no-dup (M @ M')
and a2: atms-of D ⊆ atm-of ' lits-of M'
shows ∀ x ∈ atms-of D. x ∉ atm-of ' lits-of M
proof -
{ fix aa :: 'a
  have ff1: ∧ l ms. undefined-lit ms l ∨ atm-of l
    ∈ set (map (λm. atm-of (lit-of (m::('a, 'b, 'c) marked-lit))) ms)
    by (simp add: defined-lit-map)
  have ff2: ∧ a. a ∉ atms-of D ∨ a ∈ atm-of ' lits-of M'
    using a2 by (meson subsetCE)
  have ff3: ∧ a. a ∉ set (map (λm. atm-of (lit-of m)) M')
    ∨ a ∉ set (map (λm. atm-of (lit-of m)) M)
    using a1 by (metis (lifting) IntI distinct-append empty-iff map-append)
  have ∀ L a f. ∃ l. ((a::'a) ∉ f ' L ∨ (l::'a literal) ∈ L) ∧ (a ∉ f ' L ∨ f l = a)
    by blast
  then have aa ∉ atms-of D ∨ aa ∉ atm-of ' lits-of M
    using ff3 ff2 ff1 by (metis (no-types) Marked-Propagated-in-iff-in-lits-of) }
then show ?thesis
  by blast
qed

```

**lemma** *cdcl<sub>W</sub>-propagate-is-conclusion:*

```

assumes
  cdclW S S' and
  inv: cdclW-M-level-inv S and
  decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
  learned: cdclW-learned-clause S and
  confl: ∀ T. conflicting S = C-Clause T ⟶ trail S ⊨as CNot T and
  alien: no-strange-atm S
shows all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
case restart
then show ?case by auto
next
case forget
then show ?case using decomp by auto
next
case conflict
then show ?case using decomp by auto
next
case (resolve L C M D)
note tr = this(1) and T = this(4)

```

```

let ?decomp = get-all-marked-decomposition M
have M: set ?decomp = insert (hd ?decomp) (set (tl ?decomp))
  by (cases ?decomp) auto
show ?case
  using decomp tr T unfolding all-decomposition-implies-def
  by (cases hd (get-all-marked-decomposition M))
    (auto simp: M)
next
case (skip L C' M D) note tr = this(1) and T = this(5)
have M: set (get-all-marked-decomposition M)
  = insert (hd (get-all-marked-decomposition M)) (set (tl (get-all-marked-decomposition M)))
  by (cases get-all-marked-decomposition M) auto
show ?case
  using decomp tr T unfolding all-decomposition-implies-def
  by (cases hd (get-all-marked-decomposition M))
    (auto simp add: M)
next
case decide note S = this(1) and undef = this(2) and T = this(4)
show ?case using decomp T undef unfolding S all-decomposition-implies-def by auto
next
case (propagate C L T) note propa = this(2) and undef = this(3) and T = this(5)
obtain a y where ay: hd (get-all-marked-decomposition (trail S)) = (a, y)
  by (cases hd (get-all-marked-decomposition (trail S)))
then have M: trail S = y @ a using get-all-marked-decomposition-decomp by blast
have M': set (get-all-marked-decomposition (trail S))
  = insert (a, y) (set (tl (get-all-marked-decomposition (trail S))))
  using ay by (cases get-all-marked-decomposition (trail S)) auto
have (λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set y
  using decomp ay unfolding all-decomposition-implies-def
  by (cases get-all-marked-decomposition (trail S)) fastforce+
then have a-Un-N-M: (λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S)
  ⊨ps (λa. {#lit-of a#}) ' set (trail S)
  unfolding M by (auto simp add: all-in-true-clss-clss image-Un)

have (λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S) ⊨p {#L#} (is ?I ⊨p -)
proof (rule true-clss-clss-plus-CNot)
  show ?I ⊨p C + {#L#}
  using propa propagate.premis learned confl unfolding M
  by (metis Un-iff cdclW-learned-clause-def clauses-def mem-set-mset-iff propagate.hyps(1)
    set-mset-union true-clss-clss-in-imp-true-clss-clss true-clss-clss-mono-l2
    union-trus-clss-clss)
next
have (λm. {#lit-of m#}) ' set (trail S) ⊨ps CNot C
  using ⟨(trail S) ⊨as CNot C⟩ true-annots-true-clss-clss by blast
then show ?I ⊨ps CNot C
  using a-Un-N-M true-clss-clss-left-right true-clss-clss-union-l-r by blast
qed
moreover have ∧aa b.
  ∀ (Ls, seen) ∈ set (get-all-marked-decomposition (y @ a)).
    (λa. {#lit-of a#}) ' set Ls ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set seen
  ⇒ (aa, b) ∈ set (tl (get-all-marked-decomposition (y @ a)))
  ⇒ (λa. {#lit-of a#}) ' set aa ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set b
  by (metis (no-types, lifting) case-prod-conv get-all-marked-decomposition-never-empty-sym
    list.collapse list.set-intros(2))

```

```

ultimately show ?case
  using decomp T undef unfolding ay all-decomposition-implies-def
  using M  $\langle \lambda a. \{\#lit\text{-of } a\# \} \rangle$  ' set  $a \cup \text{set-mset } (\text{init-clss } S) \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ' set  $y$ 
  ay by auto
next
case (backtrack K i M1 M2 L D T) note decomp' = this(1) and lev-L = this(2) and conf = this(3)
and
  undef = this(6) and T = this(7)
have  $\forall l \in \text{set } M2. \neg \text{is-marked } l$ 
  using get-all-marked-decomposition-snd-not-marked backtrack.hyps(1) by blast
obtain M0 where M: trail  $S = M0 @ M2 @ \text{Marked } K (i + 1) \# M1$ 
  using decomp' by auto
show ?case unfolding all-decomposition-implies-def
proof
  fix x
  assume  $x \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } T))$ 
  then have x:  $x \in \text{set } (\text{get-all-marked-decomposition } (\text{Propagated } L ((D + \{\#L\# \})) \# M1))$ 
    using T decomp' undef inv by (simp add: cdclW-M-level-inv-decomp)
  let ?m = get-all-marked-decomposition (Propagated L ((D + {#L#})) # M1)
  let ?hd = hd ?m
  let ?tl = tl ?m
  have  $x = ?hd \vee x \in \text{set } ?tl$ 
    using x by (case-tac ?m) auto
  moreover {
    assume  $x \in \text{set } ?tl$ 
    then have  $x \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
      using tl-get-all-marked-decomposition-skip-some[of x] by (simp add: list.set-sel(2) M)
    then have case x of (Ls, seen)  $\Rightarrow (\lambda a. \{\#lit\text{-of } a\# \})$  ' set Ls
       $\cup \text{set-mset } (\text{init-clss } (T))$ 
       $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ' set seen
    using decomp learned decomp confl alien inv T undef M
    unfolding all-decomposition-implies-def cdclW-M-level-inv-def
    by auto
  }
  moreover {
    assume  $x = ?hd$ 
    obtain M1' M1'' where M1:  $\text{hd } (\text{get-all-marked-decomposition } M1) = (M1', M1'')$ 
      by (cases hd (get-all-marked-decomposition M1))
    then have x':  $x = (M1', \text{Propagated } L ( (D + \{\#L\# \})) \# M1'')$ 
      using  $\langle x = ?hd \rangle$  by auto
    have  $(M1', M1'') \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
      using M1[symmetric] hd-get-all-marked-decomposition-skip-some[OF M1[symmetric],
        of M0 @ M2 - i + 1] unfolding M by fastforce
    then have 1:  $(\lambda a. \{\#lit\text{-of } a\# \})$  ' set  $M1' \cup \text{set-mset } (\text{init-clss } S)$ 
       $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ' set  $M1''$ 
      using decomp unfolding all-decomposition-implies-def by auto
    moreover
      have trail S  $\models_{as} CNot D$  using conf confl by auto
      then have vars-of-D:  $\text{atms-of } D \subseteq \text{atm-of } \text{' lits-of } (\text{trail } S)$ 
        unfolding atms-of-def
        by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)
      have vars-of-D:  $\text{atms-of } D \subseteq \text{atm-of } \text{' lits-of } M1$ 
        using backtrack-atms-of-D-in-M1[of S M1 L D i K M2 T] backtrack inv conf confl
        by (auto simp: cdclW-M-level-inv-decomp)
      have no-dup (trail S) using inv by (auto simp: cdclW-M-level-inv-decomp)
  }

```

```

then have vars-in-M1:
   $\forall x \in \text{atms-of } D. x \notin \text{atm-of } \langle \text{lits-of } (M0 @ M2 @ \text{Marked } K (i + 1) \# []) \rangle$ 
  using vars-of-D distinct-atms-of-incl-not-in-other[of  $M0 @ M2 @ \text{Marked } K (i + 1) \# []$ 
     $M1$ ]
  unfolding  $M$  by auto
have  $M1 \models_{as} CNot D$ 
  using vars-in-M1 true-annots-remove-if-notin-vars[of  $M0 @ M2 @ \text{Marked } K (i + 1) \# []$ 
     $M1 CNot D$ ] ⟨trail S ⟩  $\models_{as} CNot D$  unfolding  $M$  lits-of-def by simp
have  $M1 = M1'' @ M1'$  by (simp add: M1 get-all-marked-decomposition-decomp)
have  $TT: (\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } M1' \cup \text{set-mset } (init\text{-clss } S) \rangle \models_{ps} CNot D$ 
  using true-annots-true-clss-cl[OF  $\langle M1 \models_{as} CNot D \rangle$ ] true-clss-clss-left-right[OF 1,
    of  $CNot D$ ] unfolding  $\langle M1 = M1'' @ M1' \rangle$  by (auto simp add: inf-sup-aci(5,7))
have  $init\text{-clss } S \models_{pm} D + \{ \#L \# \}$ 
  using conf learned cdclW-learned-clause-def confl by blast
then have  $T': (\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } M1' \cup \text{set-mset } (init\text{-clss } S) \rangle \models_p D + \{ \#L \# \}$  by auto
have  $\text{atms-of } (D + \{ \#L \# \}) \subseteq \text{atms-of-mu } (clauses S)$ 
  using alien conf unfolding no-strange-atm-def clauses-def by auto
then have  $(\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } M1' \cup \text{set-mset } (init\text{-clss } S) \rangle \models_p \{ \#L \# \}$ 
  using true-clss-clss-plus-CNot[OF  $T' TT$ ] by auto
ultimately
  have case x of (Ls, seen)  $\Rightarrow$   $(\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } Ls \cup \text{set-mset } (init\text{-clss } T) \rangle$ 
     $\models_{ps} (\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } seen \rangle$  using  $T' T \text{ decomp}' \text{ undef inv}$  unfolding  $x'$ 
    by (simp add: cdclW-M-level-inv-decomp)
}
ultimately show case x of (Ls, seen)  $\Rightarrow$   $(\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } Ls \cup \text{set-mset } (init\text{-clss } T) \rangle$ 
   $\models_{ps} (\lambda a. \{ \#lit\text{-of } a \# \}) \langle \text{set } seen \rangle$  using  $T$  by auto
qed
qed

```

**lemma** *cdcl<sub>W</sub>-propagate-is-false*:

```

assumes
  cdclW S S' and
  lev: cdclW-M-level-inv S and
  learned: cdclW-learned-clause S and
  decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
  confl:  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot T$  and
  alien: no-strange-atm S and
  mark-confl: every-mark-is-a-conflict S
shows every-mark-is-a-conflict S'
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
case (propagate C L T) note undef = this(3) and T = this(5)
show ?case
proof (intro allI impI)
  fix  $L' \text{ mark } a \text{ b}$ 
  assume  $a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } T$ 
  then have  $(a = [] \wedge L = L' \wedge \text{mark} = C + \{ \#L \# \} \wedge b = \text{trail } S)$ 
     $\vee \text{tl } a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } S$ 
    using  $T \text{ undef}$  by (cases a) fastforce+
  moreover {
    assume  $\text{tl } a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } S$ 
    then have  $b \models_{as} CNot (\text{mark} - \{ \#L' \# \}) \wedge L' \in \# \text{ mark}$ 
    using mark-confl by auto
  }

```



```

    moreover {
      assume  $a = []$  and  $L = L'$  and  $mark = C + \{\#L\# \}$  and  $b = \text{trail } S$ 
      then have  $b \models_{as} CNot (mark - \{\#L\# \}) \wedge L \in \# mark$ 
      using  $\langle \text{trail } S \models_{as} CNot C \rangle$  by auto
    }
    ultimately show  $b \models_{as} CNot (mark - \{\#L'\# \}) \wedge L' \in \# mark$  by blast
  qed
next
case (decide L) note undef[simp] = this(2) and T = this(4)
have  $\bigwedge a La mark b. a @ Propagated La mark \# b = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S$ 
 $\implies tl a @ Propagated La mark \# b = \text{trail } S$  by (case-tac a, auto)
then show ?case using mark-confl T unfolding decide.hyps(1) by fastforce
next
case (skip L C' M D T) note tr = this(1) and T = this(5)
show ?case
proof (intro allI impI)
  fix L' mark a b
  assume  $a @ Propagated L' mark \# b = \text{trail } T$ 
  then have  $a @ Propagated L' mark \# b = M$  using tr T by simp
  then have  $(Propagated L C' \# a) @ Propagated L' mark \# b = Propagated L C' \# M$  by auto
  moreover have  $\forall La mark a b. a @ Propagated La mark \# b = Propagated L C' \# M$ 
 $\longrightarrow b \models_{as} CNot (mark - \{\#La\# \}) \wedge La \in \# mark$ 
  using mark-confl unfolding skip.hyps(1) by simp
  ultimately show  $b \models_{as} CNot (mark - \{\#L'\# \}) \wedge L' \in \# mark$  by blast
qed
next
case (conflict D)
then show ?case using mark-confl by simp
next
case (resolve L C M D T) note tr-S = this(1) and T = this(4)
show ?case unfolding resolve.hyps(1)
proof (intro allI impI)
  fix L' mark a b
  assume  $a @ Propagated L' mark \# b = \text{trail } T$ 
  then have  $Propagated L (C + \{\#L\# \}) \# M$ 
 $= (Propagated L (C + \{\#L\# \}) \# a) @ Propagated L' mark \# b$ 
  using T tr-S by auto
  then show  $b \models_{as} CNot (mark - \{\#L'\# \}) \wedge L' \in \# mark$ 
  using mark-confl unfolding resolve.hyps(1) by presburger
qed
next
case restart
then show ?case by auto
next
case forget
then show ?case using mark-confl by auto
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and conf = this(3) and undef = this(6)
and
T = this(7)
have  $\forall l \in \text{set } M2. \neg \text{is-marked } l$ 
using get-all-marked-decomposition-snd-not-marked backtrack.hyps(1) by blast
obtain M0 where  $M: \text{trail } S = M0 @ M2 @ \text{Marked } K (i + 1) \# M1$ 
using backtrack.hyps(1) by auto
have [simp]:  $\text{trail } (\text{reduce-trail-to } M1 (\text{add-learned-cls } (D + \{\#L\# \}))$ 

```

```

(update-backtrack-lvl i (update-conflicting C-True S))) = M1
using decomp lev by (auto simp: cdclW-M-level-inv-decomp)
show ?case
proof (intro allI impI)
  fix La mark a b
  assume a @ Propagated La mark # b = trail T
  then have (a = [] ∧ Propagated La mark = Propagated L (D + {#L#}) ∧ b = M1)
    ∨ tl a @ Propagated La mark # b = M1
    using M T decomp undef by (cases a) (auto)
  moreover {
    assume A: a = [] and
      P: Propagated La mark = Propagated L ( (D + {#L#})) and
      b: b = M1
    have trail S ⊨as CNot D using conf confl by auto
    then have vars-of-D: atms-of D ⊆ atm-of ' lits-of (trail S)
      unfolding atms-of-def
      by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)
    have vars-of-D: atms-of D ⊆ atm-of ' lits-of M1
      using backtrack-atms-of-D-in-M1[of S M1 L D i K M2 T] T backtrack lev confl by auto
    have no-dup (trail S) using lev by (auto simp: cdclW-M-level-inv-decomp)
    then have vars-in-M1: ∀ x ∈ atms-of D. x ∉
      atm-of ' lits-of (M0 @ M2 @ Marked K (i + 1) # [])
      using vars-of-D distinct-atms-of-incl-not-in-other[of M0 @ M2 @ Marked K (i + 1) # []
        M1] unfolding M by auto
    have M1 ⊨as CNot D
      using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # [] M1
        CNot D] (trail S ⊨as CNot D) unfolding M lits-of-def by simp
    then have b ⊨as CNot (mark - {#La#}) ∧ La ∈ # mark
      using P b by auto
  }
  moreover {
    assume tl a @ Propagated La mark # b = M1
    then obtain c' where c' @ Propagated La mark # b = trail S unfolding M by auto
    then have b ⊨as CNot (mark - {#La#}) ∧ La ∈ # mark
      using mark-confl by blast
  }
  ultimately show b ⊨as CNot (mark - {#La#}) ∧ La ∈ # mark by fast
qed
qed

```

**lemma** cdcl<sub>W</sub>-conflicting-is-false:

```

assumes
  cdclW S S' and
  M-lev: cdclW-M-level-inv S and
  confl-inv: ∀ T. conflicting S = C-Clause T ⟶ trail S ⊨as CNot T and
  marked-confl: ∀ L mark a b. a @ Propagated L mark # b = (trail S)
    ⟶ (b ⊨as CNot (mark - {#L#}) ∧ L ∈ # mark) and
  dist: distinct-cdclW-state S
shows ∀ T. conflicting S' = C-Clause T ⟶ trail S' ⊨as CNot T
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
  case (skip L C' M D) note tr-S = this(1) and T = this(5)
  then have Propagated L C' # M ⊨as CNot D using assms skip by auto
  moreover
    have L ∉ # D

```

```

proof (rule ccontr)
  assume  $\neg$  ?thesis
  then have  $- L \in \text{ lits-of } M$ 
    using in-CNot-implies-uminus(2)[of  $D \ L \ \text{Propagated } L \ C' \ \# \ M$ ]
     $\langle \text{Propagated } L \ C' \ \# \ M \models_{\text{as}} \text{CNot } D \rangle$  by simp
  then show False
    by (metis M-lev cdclW-M-level-inv-decomp(1) consistent-interp-def insert-iff
      lits-of-cons marked-lit.sel(2) skip.hyps(1))
  qed
ultimately show ?case
  using skip.hyps(1-3) true-annots-CNot-lit-of-notin-skip  $T$  unfolding cdclW-M-level-inv-def
  by fastforce
next
case (resolve  $L \ C \ M \ D \ T$ ) note  $tr = \text{this}(1)$  and  $\text{confl} = \text{this}(2)$  and  $T = \text{this}(4)$ 
show ?case
proof (intro allI impI)
  fix  $T'$ 
  have  $tl \ (\text{trail } S) \models_{\text{as}} \text{CNot } C$  using  $tr \ \text{assms}(4)$  by fastforce
  moreover
    have  $\text{distinct-mset } (D + \{\#- L\# \})$  using  $\text{confl } \text{dist}$ 
    unfolding distinct-cdclW-state-def by auto
    then have  $-L \notin \# \ D$  unfolding distinct-mset-def by auto
    have  $M \models_{\text{as}} \text{CNot } D$ 
    proof -
      have  $\text{Propagated } L \ ( (C + \{\#L\# \}) ) \ \# \ M \models_{\text{as}} \text{CNot } D \cup \text{CNot } \{\#- L\# \}$ 
      using  $\text{confl } tr \ \text{confl-inv}$  by force
      then show ?thesis
        using M-lev  $\langle -L \notin \# \ D \rangle \ tr \ \text{true-annots-lit-of-notin-skip}$ 
        unfolding cdclW-M-level-inv-def by force
    qed
    moreover assume  $\text{conflicting } T = C\text{-Clause } T'$ 
  ultimately
    show  $\text{trail } T \models_{\text{as}} \text{CNot } T'$ 
    using  $tr \ T$  by auto
  qed
qed (auto simp: assms(2) cdclW-M-level-inv-decomp)

```

**lemma** cdcl<sub>W</sub>-conflicting-decomp:

```

assumes cdclW-conflicting  $S$ 
shows  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$ 
and  $\forall L \ \text{mark } a \ b. a \ @ \ \text{Propagated } L \ \text{mark } \# \ b = (\text{trail } S)$ 
 $\longrightarrow (b \models_{\text{as}} \text{CNot } (\text{mark} - \{\#L\# \}) \wedge L \in \# \ \text{mark})$ 
using assms unfolding cdclW-conflicting-def by blast+

```

**lemma** cdcl<sub>W</sub>-conflicting-decomp2:

```

assumes cdclW-conflicting  $S$  and  $\text{conflicting } S = C\text{-Clause } T$ 
shows  $\text{trail } S \models_{\text{as}} \text{CNot } T$ 
using assms unfolding cdclW-conflicting-def by blast+

```

**lemma** cdcl<sub>W</sub>-conflicting-decomp2':

```

assumes
  cdclW-conflicting  $S$  and
   $\text{conflicting } S = C\text{-Clause } D$ 
shows  $\text{trail } S \models_{\text{as}} \text{CNot } D$ 
using assms unfolding cdclW-conflicting-def by auto

```

**lemma** *cdcl<sub>W</sub>-conflicting-S0-cdcl<sub>W</sub>[simp]*:  
*cdcl<sub>W</sub>-conflicting* (*init-state* *N*)  
**unfolding** *cdcl<sub>W</sub>-conflicting-def* **by** *auto*

#### 17.4.9 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-marked-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-marked-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** –  
**show** *S1: all-decomposition-implies-m* (*init-clss* *S'*) (*get-all-marked-decomposition* (*trail* *S'*))  
**using** *cdcl<sub>W</sub>-propagate-is-conclusion*[*OF cdcl<sub>W</sub> 4 1 2 - 5*] *8* **unfolding** *cdcl<sub>W</sub>-conflicting-def*  
**by** *blast*  
**show** *S2: cdcl<sub>W</sub>-learned-clause* *S'* **using** *cdcl<sub>W</sub>-learned-clss*[*OF cdcl<sub>W</sub> 2 4*] .  
**show** *S4: cdcl<sub>W</sub>-M-level-inv* *S'* **using** *cdcl<sub>W</sub>-consistent-inv*[*OF cdcl<sub>W</sub> 4*] .  
**show** *S5: no-strange-atm* *S'* **using** *cdcl<sub>W</sub>-no-strange-atm-inv*[*OF cdcl<sub>W</sub> 5 4*] .  
**show** *S7: distinct-cdcl<sub>W</sub>-state* *S'* **using** *distinct-cdcl<sub>W</sub>-state-inv*[*OF cdcl<sub>W</sub> 4 7*] .  
**show** *S8: cdcl<sub>W</sub>-conflicting* *S'*  
**using** *cdcl<sub>W</sub>-conflicting-is-false*[*OF cdcl<sub>W</sub> 4 - - 7*] *8* *cdcl<sub>W</sub>-propagate-is-false*[*OF cdcl<sub>W</sub> 4 2 1 - 5*]  
**unfolding** *cdcl<sub>W</sub>-conflicting-def* **by** *fast*  
**qed**

**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-marked-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-marked-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'*  
**using** *assms*  
**proof** (*induct* rule: *rtrancp-induct*)  
**case** *base*  
**case** *1* **then** **show** *?case* **by** *blast*  
**case** *2* **then** **show** *?case* **by** *blast*

```

    case 3 then show ?case by blast
    case 4 then show ?case by blast
    case 5 then show ?case by blast
    case 6 then show ?case by blast
next
case (step S' S'') note H = this
  case 1 with H(3-7)[OF this(1-6)] show ?case using cdclW-all-inv[OF H(2)]
    H by presburger
  case 2 with H(3-7)[OF this(1-6)] show ?case using cdclW-all-inv[OF H(2)]
    H by presburger
  case 3 with H(3-7)[OF this(1-6)] show ?case using cdclW-all-inv[OF H(2)]
    H by presburger
  case 4 with H(3-7)[OF this(1-6)] show ?case using cdclW-all-inv[OF H(2)]
    H by presburger
  case 5 with H(3-7)[OF this(1-6)] show ?case using cdclW-all-inv[OF H(2)]
    H by presburger
  case 6 with H(3-7)[OF this(1-6)] show ?case using cdclW-all-inv[OF H(2)]
    H by presburger
qed

lemma all-invariant-S0-cdclW:
  assumes distinct-mset-mset N
  shows all-decomposition-implies-m (init-clss (init-state N))
    (get-all-marked-decomposition (trail (init-state N)))
  and cdclW-learned-clause (init-state N)
  and  $\forall T. \text{conflicting } (init-state N) = C\text{-Clause } T \longrightarrow (trail (init-state N)) \models_{as} C\text{Not } T$ 
  and no-strange-atm (init-state N)
  and consistent-interp (lits-of (trail (init-state N)))
  and  $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = \text{trail } (init-state N) \longrightarrow$ 
     $(b \models_{as} C\text{Not } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$ 
  and distinct-cdclW-state (init-state N)
  using assms by auto

lemma cdclW-only-propagated-vars-unsat:
  assumes
    marked:  $\forall x \in \text{set } M. \neg \text{is-marked } x$  and
    DN:  $D \in \# \text{ clauses } S$  and
    D:  $M \models_{as} C\text{Not } D$  and
    inv: all-decomposition-implies-m N (get-all-marked-decomposition M) and
    state:  $\text{state } S = (M, N, U, k, C)$  and
    learned-cl: cdclW-learned-clause S and
    atm-incl: no-strange-atm S
  shows unsatisfiable (set-mset N)
proof (rule ccontr)
  assume  $\neg \text{unsatisfiable } (set-mset N)$ 
  then obtain I where
    I:  $I \models_s \text{set-mset } N$  and
    cons: consistent-interp I and
    tot: total-over-m I (set-mset N)
  unfolding satisfiable-def by auto
  have atms-of-mu  $N \cup \text{atms-of-mu } U = \text{atms-of-mu } N$ 
  using atm-incl state unfolding total-over-m-def no-strange-atm-def
  by (auto simp add: clauses-def)
  then have total-over-m I (set-mset N) using tot unfolding total-over-m-def by auto

```

**moreover** have  $N \models_{psm} U$  **using** *learned-cl state unfolding cdcl<sub>W</sub>-learned-clause-def* **by** *auto*  
**ultimately** have  $I-D: I \models D$   
**using**  $I \ DN \ cons \ state \ unfolding \ true-clss-clss-def \ true-clss-def \ Ball-def$   
**by** (*metis Un-iff ‹atms-of-mu N  $\cup$  atms-of-mu U = atms-of-mu N› atms-of-m-union clauses-def mem-set-mset-iff prod.inject set-mset-union total-over-m-def*)  
  
**have**  $l0: \{\{\#lit-of\ L\# \} \mid L. \ is-marked\ L \wedge L \in \ set\ M\} = \{\}$  **using** *marked* **by** *auto*  
**have**  $atms-of-m \ (set-mset\ N \cup (\lambda a. \ \{\#lit-of\ a\# \}) \ ' \ set\ M) = atms-of-mu\ N$   
**using** *atm-incl state unfolding no-strange-atm-def* **by** *auto*  
**then** have  $total-over-m\ I \ (set-mset\ N \cup (\lambda a. \ \{\#lit-of\ a\# \}) \ ' \ (set\ M))$   
**using** *tot unfolding total-over-m-def* **by** *auto*  
**then** have  $I \models_s (\lambda a. \ \{\#lit-of\ a\# \}) \ ' \ (set\ M)$   
**using** *all-decomposition-implies-propagated-lits-are-implied[OF inv] cons I*  
**unfolding** *true-clss-clss-def l0* **by** *auto*  
**then** have  $IM: I \models_s (\lambda a. \ \{\#lit-of\ a\# \}) \ ' \ set\ M$  **by** *auto*  
**{**  
**fix**  $K$   
**assume**  $K \in \# \ D$   
**then** have  $-K \in \ lits-of\ M$   
**using**  $D$  **unfolding** *true-annots-def Ball-def CNot-def true-annot-def true-clss-def true-lit-def*  
*Bex-mset-def* **by** (*metis (mono-tags, lifting) count-single less-not-refl mem-Collect-eq*)  
**then** have  $-K \in I$  **using**  $IM$  *true-clss-singleton-lit-of-implies-incl lits-of-def* **by** *fastforce*  
**}**  
**then** have  $\neg I \models D$  **using** *cons unfolding true-clss-def true-lit-def consistent-interp-def* **by** *auto*  
**then** show *False* **using**  $I-D$  **by** *blast*  
**qed**

We have actually a much stronger theorem, namely *all-decomposition-implies ?N (get-all-marked-decomposition ?M)  $\implies ?N \cup \{\{\#lit-of\ L\# \} \mid L. \ is-marked\ L \wedge L \in \ set\ ?M\} \models_{ps} (\lambda a. \ \{\#lit-of\ a\# \}) \ ' \ set\ ?M$* , that show that the only choices we made are marked in the formula

**lemma**  
**assumes** *all-decomposition-implies-m N (get-all-marked-decomposition M)*  
**and**  $\forall m \in \ set\ M. \neg is-marked\ m$   
**shows**  $set-mset\ N \models_{ps} (\lambda a. \ \{\#lit-of\ a\# \}) \ ' \ set\ M$   
**proof** –  
**have**  $T: \{\{\#lit-of\ L\# \} \mid L. \ is-marked\ L \wedge L \in \ set\ M\} = \{\}$  **using** *assms(2)* **by** *auto*  
**then** show *?thesis*  
**using** *all-decomposition-implies-propagated-lits-are-implied[OF assms(1)] unfolding T* **by** *simp*  
**qed**

**lemma** *conflict-with-false-implies-unsat:*

**assumes**  
 $cdcl_W: cdcl_W\ S\ S' \text{ and}$   
 $lev: cdcl_W\text{-}M\text{-level-inv}\ S \text{ and}$   
 $[simp]: conflicting\ S' = C\text{-}Clause\ \{\#\} \text{ and}$   
 $learned: cdcl_W\text{-}learned\text{-}clause\ S$   
**shows** *unsatisfiable (set-mset (init-clss S))*  
**using** *assms*  
**proof** –  
**have**  $cdcl_W\text{-}learned\text{-}clause\ S' \text{ using } cdcl_W\text{-}learned\text{-}clss\ cdcl_W\ learned\ lev$  **by** *auto*  
**then** have  $init-clss\ S' \models_{pm} \{\#\}$  **using** *assms(3) unfolding cdcl\_W-learned-clause-def* **by** *auto*  
**then** have  $init-clss\ S \models_{pm} \{\#\}$   
**using**  $cdcl_W\text{-}init-clss[OF\ assms(1)\ lev]$  **by** *auto*  
**then** show *?thesis* **unfolding** *satisfiable-def true-clss-clss-def* **by** *auto*

qed

**lemma** *conflict-with-false-implies-terminated*:

**assumes**  $cdcl_W S S'$   
**and** *conflicting*  $S = C\text{-Clause } \{\#\}$   
**shows** *False*  
**using** *assms* **by** (*induct rule: cdcl<sub>W</sub>-all-induct*) *auto*

#### 17.4.10 No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

**lemma** *learned-clss-are-not-tautologies*:

**assumes**  
 $cdcl_W S S'$  **and**  
 $lev: cdcl_W\text{-}M\text{-level-inv } S$  **and**  
*conflicting*:  $cdcl_W\text{-conflicting } S$  **and**  
 $no\text{-tauto}: \forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$   
**shows**  $\forall s \in \# \text{ learned-clss } S'. \neg \text{tautology } s$   
**using** *assms*

**proof** (*induct rule: cdcl<sub>W</sub>-all-induct-lev2*)

**case** (*backtrack*  $K i M1 M2 L D$ ) **note**  $confl = this(3)$

**have** *consistent-interp* (*lits-of* (*trail*  $S$ )) **using**  $lev$  **by** (*auto simp: cdcl<sub>W</sub>-M-level-inv-decomp*)

**moreover**

**have**  $trail S \models_{as} CNot (D + \{\#L\# \})$

**using** *conflicting confl unfolding cdcl<sub>W</sub>-conflicting-def* **by** *auto*

**then have**  $lits\text{-of } (trail S) \models_s CNot (D + \{\#L\# \})$  **using** *true-annots-true-clss* **by** *blast*

**ultimately have**  $\neg \text{tautology } (D + \{\#L\# \})$  **using** *consistent-CNot-not-tautology* **by** *blast*

**then show** *?case* **using** *backtrack no-tauto*

**by** (*auto simp: cdcl<sub>W</sub>-M-level-inv-decomp split: split-if-asm*)

**next**

**case** *restart*

**then show** *?case* **using** *learned-clss-restart-state state-eq-learned-clss no-tauto*

**by** (*metis (no-types, lifting) ball-msetE ball-msetI mem-set-mset-iff set-mset-mono subsetCE*)

qed *auto*

**definition** *final-cdcl<sub>W</sub>-state* ( $S:: 'st$ )

$\longleftrightarrow (trail S \models_{asm} init\text{-clss } S$   
 $\vee ((\forall L \in set (trail S). \neg is\text{-marked } L) \wedge$   
 $(\exists C \in \# init\text{-clss } S. trail S \models_{as} CNot C)))$

**definition** *termination-cdcl<sub>W</sub>-state* ( $S:: 'st$ )

$\longleftrightarrow (trail S \models_{asm} init\text{-clss } S$   
 $\vee ((\forall L \in atms\text{-of-mu } (init\text{-clss } S). L \in atm\text{-of } 'lits\text{-of } (trail S))$   
 $\wedge (\exists C \in \# init\text{-clss } S. trail S \models_{as} CNot C)))$

### 17.5 CDCL Strong Completeness

**fun** *mapi* ::  $('a \Rightarrow nat \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list}$  **where**

*mapi* - -  $[] = []$  |

*mapi*  $f n (x \# xs) = f x n \# mapi f (n - 1) xs$

**lemma** *mark-not-in-set-mapi[simp]*:  $L \notin set M \Longrightarrow Marked L k \notin set (mapi Marked i M)$

**by** (*induct M arbitrary: i*) *auto*

**lemma** *propagated-not-in-set-mapi[simp]*:  $L \notin \text{set } M \implies \text{Propagated } L \notin \text{set } (\text{mapi } \text{Marked } i \text{ } M)$   
**by** (*induct*  $M$  *arbitrary*:  $i$ ) *auto*

**lemma** *image-set-mapi*:  
 $f \text{ ' set } (\text{mapi } g \text{ } i \text{ } M) = \text{set } (\text{mapi } (\lambda x \text{ } i. f \text{ } (g \text{ } x \text{ } i)) \text{ } i \text{ } M)$   
**by** (*induction*  $M$  *arbitrary*:  $i$ ) *auto*

**lemma** *mapi-map-convert*:  
 $\forall x \text{ } i \text{ } j. f \text{ } x \text{ } i = f \text{ } x \text{ } j \implies \text{mapi } f \text{ } i \text{ } M = \text{map } (\lambda x. f \text{ } x \text{ } 0) \text{ } M$   
**by** (*induction*  $M$  *arbitrary*:  $i$ ) *auto*

**lemma** *defined-lit-mapi*:  $\text{defined-lit } (\text{mapi } \text{Marked } i \text{ } M) \text{ } L \longleftrightarrow \text{atm-of } L \in \text{atm-of ' set } M$   
**by** (*induction*  $M$ ) (*auto simp*: *defined-lit-map image-set-mapi mapi-map-convert*)

**lemma** *cdcl<sub>W</sub>-can-do-step*:  
**assumes**  
*consistent-interp* (*set*  $M$ ) **and**  
*distinct*  $M$  **and**  
 $\text{atm-of ' } (\text{set } M) \subseteq \text{atms-of-mu } N$   
**shows**  $\exists S. \text{rtrancp } \text{cdcl}_W \text{ } (\text{init-state } N) \text{ } S$   
 $\wedge \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) \text{ } M, N, \{\#\}, \text{length } M, C\text{-True})$   
**using** *assms*  
**proof** (*induct*  $M$ )  
**case** *Nil*  
**then show** *?case* **by** *auto*  
**next**  
**case** (*Cons*  $L \text{ } M$ ) **note**  $IH = \text{this}(1)$   
**have** *consistent-interp* (*set*  $M$ ) **and** *distinct*  $M$  **and**  $\text{atm-of ' set } M \subseteq \text{atms-of-mu } N$   
**using** *Cons.prem*s(1–3) **unfolding** *consistent-interp-def* **by** *auto*  
**then obtain**  $S$  **where**  
 $st: \text{cdcl}_W^{**} \text{ } (\text{init-state } N) \text{ } S$  **and**  
 $S: \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) \text{ } M, N, \{\#\}, \text{length } M, C\text{-True})$   
**using**  $IH$  **by** *auto*  
**let**  $?S_0 = \text{incr-lvl } (\text{cons-trail } (\text{Marked } L \text{ } (\text{length } M + 1)) \text{ } S)$   
**have** *undefined-lit* (*mapi* *Marked* (*length*  $M$ )  $M$ )  $L$   
**using** *Cons.prem*s(1,2) **unfolding** *defined-lit-def consistent-interp-def* **by** *fastforce*  
**moreover have** *init-clss*  $S = N$   
**using**  $S$  **by** *blast*  
**moreover have**  $\text{atm-of } L \in \text{atms-of-mu } N$  **using** *Cons.prem*s(3) **by** *auto*  
**moreover have** *undef*: *undefined-lit* (*trail*  $S$ )  $L$   
**using**  $S \text{ } \langle \text{distinct } (L \# M) \rangle$  *calculation*(1) **by** (*auto simp*: *defined-lit-mapi defined-lit-map*)  
**ultimately have**  $\text{cdcl}_W \text{ } S \text{ } ?S_0$   
**using**  $\text{cdcl}_W.\text{other}[OF \text{ } \text{cdcl}_W.\text{o.decide}[OF \text{ } \text{decide-rule}[OF \text{ } S, \text{of } L \text{ } ?S_0]]] \text{ } S$  **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)  
**then show** *?case*  
**using**  $st \text{ } S \text{ } \text{undef}$  **by** (*auto intro*!: *exI*[*of* -  $?S_0$ ])  
**qed**

**lemma** *cdcl<sub>W</sub>-strong-completeness*:  
**assumes**  
 $\text{set } M \models_s \text{set-mset } N$  **and**  
*consistent-interp* (*set*  $M$ ) **and**  
*distinct*  $M$  **and**  
 $\text{atm-of ' } (\text{set } M) \subseteq \text{atms-of-mu } N$   
**obtains**  $S$  **where**



```

state  $S = (\text{mapi Marked } (\text{length } M) M, N, \{\#\}, \text{length } M, C\text{-True})$  and
rtrancpl cdclW (init-state  $N$ )  $S$  and
final-cdclW-state  $S$ 
proof -
  obtain  $S$  where
    st: rtrancpl cdclW (init-state  $N$ )  $S$  and
    S: state  $S = (\text{mapi Marked } (\text{length } M) M, N, \{\#\}, \text{length } M, C\text{-True})$ 
    using cdclW-can-do-step[OF assms(2-4)] by auto
  have lits-of (mapi Marked (length  $M$ )  $M$ ) = set  $M$ 
    by (induct  $M$ , auto)
  then have mapi Marked (length  $M$ )  $M \models_{asm} N$  using assms(1) true-annots-true-cls by metis
  then have final-cdclW-state  $S$ 
    using  $S$  unfolding final-cdclW-state-def by auto
  then show ?thesis using that st  $S$  by blast
qed

```

## 17.6 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

### 17.6.1 Definition

```

lemma trancpl-conflict-iff[iff]:
  full1 conflict  $S S' \longleftrightarrow$  conflict  $S S'$ 
proof -
  have trancpl conflict  $S S' \implies$  conflict  $S S'$ 
    unfolding full1-def by (induct rule: trancpl.induct) force+
  then have trancpl conflict  $S S' \implies$  conflict  $S S'$  by (meson rtrancplD)
  then show ?thesis unfolding full1-def by (metis conflictE conflicting-clause.simps(3)
    conflicting-update-conflicting state-eq-conflicting trancpl.intros(1))
qed

```

```

inductive cdclW-cp :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  conflict[intro]: conflict  $S S' \implies$  cdclW-cp  $S S'$  |
  propagate': propagate  $S S' \implies$  cdclW-cp  $S S'$ 

```

```

lemma rtrancpl-cdclW-cp-rtrancpl-cdclW:
  cdclW-cp**  $S T \implies$  cdclW**  $S T$ 
  by (induction rule: rtrancpl-induct) (auto simp: cdclW-cp.simps dest: cdclW.intros)

```

```

lemma cdclW-cp-state-eq-compatible:
  assumes
    cdclW-cp  $S T$  and
     $S \sim S'$  and
     $T \sim T'$ 
  shows cdclW-cp  $S' T'$ 
  using assms
  apply (induction)
    using conflict-state-eq-compatible apply auto[1]
  using propagate' propagate-state-eq-compatible by auto

```

```

lemma trancpl-cdclW-cp-state-eq-compatible:
  assumes
    cdclW-cp**  $S T$  and
     $S \sim S'$  and

```

```

     $T \sim T'$ 
  shows  $cdcl_W\text{-}cp^{++} S' T'$ 
  using assms
proof induction
  case base
  then show ?case
    using  $cdcl_W\text{-}cp\text{-}state\text{-}eq\text{-}compatible$  by blast
next
  case (step U V)
  obtain  $ss :: 'st$  where
     $cdcl_W\text{-}cp S ss \wedge cdcl_W\text{-}cp^{**} ss U$ 
  by (metis (no-types) step(1) tranclpD)
  then show ?case
    by (meson  $cdcl_W\text{-}cp\text{-}state\text{-}eq\text{-}compatible$   $rtranclp.rtrancl\text{-}into\text{-}rtrancl$   $rtranclp\text{-}into\text{-}tranclp2$ 
      state-eq-ref step(2) step(4) step(5))
qed

lemma conflicting-clause-full- $cdcl_W\text{-}cp$ :
   $conflicting S \neq C\text{-}True \implies full\ cdcl_W\text{-}cp S S$ 
unfolding full-def  $rtranclp\text{-}unfold$   $tranclp\text{-}unfold$  by (auto simp add:  $cdcl_W\text{-}cp.simps$ )

lemma skip-unique:
   $skip S T \implies skip S T' \implies T \sim T'$ 
  by (fastforce simp: state-eq-def simp del: state-simp)

lemma resolve-unique:
   $resolve S T \implies resolve S T' \implies T \sim T'$ 
  by (fastforce simp: state-eq-def simp del: state-simp)

lemma  $cdcl_W\text{-}cp\text{-}no\text{-}more\text{-}clauses$ :
  assumes  $cdcl_W\text{-}cp S S'$ 
  shows  $clauses S = clauses S'$ 
  using assms by (induct rule:  $cdcl_W\text{-}cp.induct$ ) (auto elim!: conflictE propagateE)

lemma  $tranclp\text{-}cdcl_W\text{-}cp\text{-}no\text{-}more\text{-}clauses$ :
  assumes  $cdcl_W\text{-}cp^{++} S S'$ 
  shows  $clauses S = clauses S'$ 
  using assms by (induct rule:  $tranclp.induct$ ) (auto dest:  $cdcl_W\text{-}cp\text{-}no\text{-}more\text{-}clauses$ )

lemma  $rtranclp\text{-}cdcl_W\text{-}cp\text{-}no\text{-}more\text{-}clauses$ :
  assumes  $cdcl_W\text{-}cp^{**} S S'$ 
  shows  $clauses S = clauses S'$ 
  using assms by (induct rule:  $rtranclp.induct$ ) (fastforce dest:  $cdcl_W\text{-}cp\text{-}no\text{-}more\text{-}clauses$ )+

lemma no-conflict-after-conflict:
   $conflict S T \implies \neg conflict T U$ 
  by fastforce

lemma no-propagate-after-conflict:
   $conflict S T \implies \neg propagate T U$ 
  by fastforce

lemma  $tranclp\text{-}cdcl_W\text{-}cp\text{-}propagate\text{-}with\text{-}conflict\text{-}or\text{-}not$ :
  assumes  $cdcl_W\text{-}cp^{++} S U$ 
  shows  $(propagate^{++} S U \wedge conflicting U = C\text{-}True)$ 

```

$\vee (\exists T D. \text{propagate}^{**} S T \wedge \text{conflict} T U \wedge \text{conflicting} U = C\text{-Clause } D)$   
**proof** –  
**have**  $\text{propagate}^{++} S U \vee (\exists T. \text{propagate}^{**} S T \wedge \text{conflict} T U)$   
**using** *assms by induction*  
*(force simp: cdcl<sub>W</sub>-cp.simps tranclp-into-rtranclp dest: no-conflict-after-conflict no-propagate-after-conflict)+*  
**moreover**  
**have**  $\text{propagate}^{++} S U \implies \text{conflicting} U = C\text{-True}$   
**unfolding** *tranclp-unfold-end by auto*  
**moreover**  
**have**  $\bigwedge T. \text{conflict} T U \implies \exists D. \text{conflicting} U = C\text{-Clause } D$   
**by** *auto*  
**ultimately show** *?thesis by meson*  
**qed**

**lemma** *cdcl<sub>W</sub>-cp-conflicting-not-empty[simp]: conflicting S = C-Clause D  $\implies \neg \text{cdcl}_W\text{-cp } S S'$*   
**proof**  
**assume** *cdcl<sub>W</sub>-cp S S' and conflicting S = C-Clause D*  
**then show** *False by (induct rule: cdcl<sub>W</sub>-cp.induct) auto*  
**qed**

**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*  
**using** *assms conflict' apply blast*  
**by** *(meson assms conflict' propagate')*

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 *full cdcl<sub>W</sub>-cp S' S''*

**inductive** *cdcl<sub>W</sub>-stgy* :: *'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where*  
*conflict': full1 cdcl<sub>W</sub>-cp S S'  $\implies$  cdcl<sub>W</sub>-stgy S S' |*  
*other': cdcl<sub>W</sub>-o 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>-stgy S S''*

## 17.6.2 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'*  
**using** *assms by (induct rule: cdcl<sub>W</sub>-cp.induct) fastforce+*

**lemma** *rtranclp-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'*  
**using** *assms by (induct rule: rtranclp-induct) (fastforce dest: cdcl<sub>W</sub>-cp-learned-clause-inv)+*

**lemma** *tranclp-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'*  
**using** *assms by (simp add: rtranclp-cdcl<sub>W</sub>-cp-learned-clause-inv tranclp-into-rtranclp)*

**lemma** *cdcl<sub>W</sub>-cp-backtrack-lvl:*  
**assumes** *cdcl<sub>W</sub>-cp S S'*  
**shows** *backtrack-lvl S = backtrack-lvl S'*

**using** *assms* **by** (*induct rule*: *cdcl<sub>W</sub>-cp.induct*) *fastforce*+

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-backtrack-lvl*:  
 assumes *cdcl<sub>W</sub>-cp<sup>\*\*</sup> S S'*  
 shows *backtrack-lvl S = backtrack-lvl S'*  
**using** *assms* **by** (*induct rule*: *rtrancpl-induct*) (*fastforce dest*: *cdcl<sub>W</sub>-cp-backtrack-lvl*)+

**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'*  
**using** *assms*  
**proof** (*induct rule*: *cdcl<sub>W</sub>-cp.induct*)  
 case (*conflict'*)  
 then show ?case **using** *cdcl<sub>W</sub>-consistent-inv cdcl<sub>W</sub>.conflict* **by** *blast*  
**next**  
 case (*propagate' S S'*)  
 have *cdcl<sub>W</sub> S S'*  
 using *propagate'.hyps(1) propagate* **by** *blast*  
 then show *cdcl<sub>W</sub>-M-level-inv S'*  
 using *propagate'.prems(1) cdcl<sub>W</sub>-consistent-inv propagate* **by** *blast*  
**qed**

**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'*  
**using** *assms* **unfolding** *full1-def*  
**proof** –  
 have *cdcl<sub>W</sub>-cp<sup>++</sup> S S'* and *cdcl<sub>W</sub>-M-level-inv S* **using** *assms* **unfolding** *full1-def* **by** *auto*  
 then show ?thesis **by** (*induct rule*: *trancpl.induct*) (*blast intro*: *cdcl<sub>W</sub>-cp-consistent-inv*)+  
**qed**

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-consistent-inv*:  
 assumes *rtrancpl 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'*  
**using** *assms* **unfolding** *full1-def*  
**by** (*induction rule*: *rtrancpl-induct*) (*blast intro*: *cdcl<sub>W</sub>-cp-consistent-inv*)+

**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'*  
**using** *assms* **apply** (*induct rule*: *cdcl<sub>W</sub>-stgy.induct*)  
**unfolding** *full-unfold* **by** (*blast intro*: *cdcl<sub>W</sub>-consistent-inv full1-cdcl<sub>W</sub>-cp-consistent-inv cdcl<sub>W</sub>.other*)+

**lemma** *rtrancpl-cdcl<sub>W</sub>-stgy-consistent-inv*:  
 assumes *cdcl<sub>W</sub>-stgy<sup>\*\*</sup> S S'*  
 and *cdcl<sub>W</sub>-M-level-inv S*  
 shows *cdcl<sub>W</sub>-M-level-inv S'*  
**using** *assms* **by** *induction* (*auto dest*!: *cdcl<sub>W</sub>-stgy-consistent-inv*)

**lemma** *cdcl<sub>W</sub>-cp-no-more-init-clss*:

**assumes**  $cdcl_W\text{-cp } S \ S'$   
**shows**  $init\text{-clss } S = init\text{-clss } S'$   
**using** *assms* **by** (*induct rule*:  $cdcl_W\text{-cp.induct}$ ) *auto*

**lemma**  $trancpl\text{-}cdcl_W\text{-cp-no-more-init-clss}$ :  
**assumes**  $cdcl_W\text{-cp}^{++} S \ S'$   
**shows**  $init\text{-clss } S = init\text{-clss } S'$   
**using** *assms* **by** (*induct rule*:  $trancpl.induct$ ) (*auto dest*:  $cdcl_W\text{-cp-no-more-init-clss}$ )

**lemma**  $cdcl_W\text{-stgy-no-more-init-clss}$ :  
**assumes**  $cdcl_W\text{-stgy } S \ S'$  **and**  $cdcl_W\text{-M-level-inv } S$   
**shows**  $init\text{-clss } S = init\text{-clss } S'$   
**using** *assms*  
**apply** (*induct rule*:  $cdcl_W\text{-stgy.induct}$ )  
**unfolding**  $full1\text{-def}$   $full\text{-def}$  **apply** (*blast dest*:  $trancpl\text{-}cdcl_W\text{-cp-no-more-init-clss}$   
 $trancpl\text{-}cdcl_W\text{-o-no-more-init-clss}$ )  
**by** (*metis*  $cdcl_W\text{-o-no-more-init-clss}$   $rtrancpl\text{-unfold}$   $trancpl\text{-}cdcl_W\text{-cp-no-more-init-clss}$ )

**lemma**  $rtrancpl\text{-}cdcl_W\text{-stgy-no-more-init-clss}$ :  
**assumes**  $cdcl_W\text{-stgy}^{**} S \ S'$  **and**  $cdcl_W\text{-M-level-inv } S$   
**shows**  $init\text{-clss } S = init\text{-clss } S'$   
**using** *assms*  
**apply** (*induct rule*:  $rtrancpl\text{-induct}$ , *simp*)  
**using**  $cdcl_W\text{-stgy-no-more-init-clss}$  **by** (*simp add*:  $rtrancpl\text{-}cdcl_W\text{-stgy-consistent-inv}$ )

**lemma**  $cdcl_W\text{-cp-dropWhile-trail'}$ :  
**assumes**  $cdcl_W\text{-cp } S \ S'$   
**obtains**  $M$  **where**  $trail \ S' = M @ trail \ S$  **and**  $(\forall l \in set \ M. \neg is\text{-marked } l)$   
**using** *assms* **by** *induction fastforce+*

**lemma**  $rtrancpl\text{-}cdcl_W\text{-cp-dropWhile-trail'}$ :  
**assumes**  $cdcl_W\text{-cp}^{**} S \ S'$   
**obtains**  $M :: ('v, nat, 'v \text{ clause}) \text{ marked-lit list}$  **where**  
 $trail \ S' = M @ trail \ S$  **and**  $\forall l \in set \ M. \neg is\text{-marked } l$   
**using** *assms* **by** *induction (fastforce dest!: cdcl\_W-cp-dropWhile-trail')+*

**lemma**  $cdcl_W\text{-cp-dropWhile-trail}$ :  
**assumes**  $cdcl_W\text{-cp } S \ S'$   
**shows**  $\exists M. trail \ S' = M @ trail \ S \wedge (\forall l \in set \ M. \neg is\text{-marked } l)$   
**using** *assms* **by** *induction fastforce+*

**lemma**  $rtrancpl\text{-}cdcl_W\text{-cp-dropWhile-trail}$ :  
**assumes**  $cdcl_W\text{-cp}^{**} S \ S'$   
**shows**  $\exists M. trail \ S' = M @ trail \ S \wedge (\forall l \in set \ M. \neg is\text{-marked } l)$   
**using** *assms* **by** *induction (fastforce dest: cdcl\_W-cp-dropWhile-trail)+*

This theorem can be seen a a termination theorem for  $cdcl_W\text{-cp}$ .

**lemma**  $length\text{-model-le-vars}$ :  
**assumes**  
 $no\text{-strange-atm } S$  **and**  
 $no\text{-d: no-dup } (trail \ S)$  **and**  
 $finite \ (atms\text{-of-mu } (init\text{-clss } S))$   
**shows**  $length \ (trail \ S) \leq card \ (atms\text{-of-mu } (init\text{-clss } S))$   
**proof** –  
**obtain**  $M \ N \ U \ k \ D$  **where**  $S: state \ S = (M, N, U, k, D)$  **by** (*cases state S, auto*)

```

have finite (atm-of ' lits-of (trail S))
  using assms(1,3) unfolding S by (auto simp add: finite-subset)
have length (trail S) = card (atm-of ' lits-of (trail S))
  using no-dup-length-eq-card-atm-of-lits-of no-d by blast
then show ?thesis using assms(1) unfolding no-strange-atm-def
  by (auto simp add: assms(3) card-mono)
qed

```

**lemma** *cdcl<sub>W</sub>-cp-decreasing-measure:*

```

assumes
  cdclW: cdclW-cp S T and
  M-lev: cdclW-M-level-inv S and
  alien: no-strange-atm S
shows (λS. card (atms-of-mu (init-clss S)) - length (trail S)
  + (if conflicting S = C-True then 1 else 0)) S
  > (λS. card (atms-of-mu (init-clss S)) - length (trail S)
  + (if conflicting S = C-True then 1 else 0)) T
using assms
proof -
  have length (trail T) ≤ card (atms-of-mu (init-clss T))
  apply (rule length-model-le-vars)
  using cdclW-no-strange-atm-inv alien M-lev apply (meson cdclW cdclW.simps cdclW-cp.cases)
  using M-lev cdclW cdclW-cp-consistent-inv cdclW-M-level-inv-def apply blast
  using cdclW by (auto simp: cdclW-cp.simps)
with assms
show ?thesis by induction (auto split: split-if-asm)+
qed

```

**lemma** *cdcl<sub>W</sub>-cp-wf:* wf {(b,a). (cdcl<sub>W</sub>-M-level-inv a ∧ no-strange-atm a) ∧ cdcl<sub>W</sub>-cp a b}

```

apply (rule wf-wf-if-measure'[of less-than - -
  (λS. card (atms-of-mu (init-clss S)) - length (trail S)
  + (if conflicting S = C-True then 1 else 0))])
  apply simp
  using cdclW-cp-decreasing-measure unfolding less-than-iff by blast

```

**lemma** *rtrancp-cdcl<sub>W</sub>-all-struct-inv-cdcl<sub>W</sub>-cp-iff-rtrancp-cdcl<sub>W</sub>-cp:*

```

assumes
  lev: cdclW-M-level-inv S and
  alien: no-strange-atm S
shows (λa b. (cdclW-M-level-inv a ∧ no-strange-atm a) ∧ cdclW-cp a b)** S T
  ⟷ cdclW-cp** S T
(is ?I S T ⟷ ?C S T)
proof
  assume
    ?I S T
  then show ?C S T by induction auto
next
  assume
    ?C S T
  then show ?I S T
  proof induction
    case base
    then show ?case by simp
  next

```

**case** (*step*  $T\ U$ ) **note**  $st = this(1)$  **and**  $cp = this(2)$  **and**  $IH = this(3)$   
**have**  $cdcl_W^{**}\ S\ T$   
    **by** (*metis* *rtranclp-unfold* *cdcl\_W-cp-conflicting-not-empty* *cp* *st*  
      *rtranclp-propagate-is-rtranclp-cdcl\_W* *trancpl-cdcl\_W-cp-propagate-with-conflict-or-not*)  
**then have**  
    *cdcl\_W-M-level-inv*  $T$  **and**  
    *no-strange-atm*  $T$   
    **using**  $\langle cdcl_W^{**}\ S\ T \rangle$  **apply** (*simp* *add: assms(1) rtranclp-cdcl\_W-consistent-inv*)  
    **using**  $\langle cdcl_W^{**}\ S\ T \rangle$  *alien* *rtranclp-cdcl\_W-no-strange-atm-inv* *lev* **by** *blast*  
**then have**  $(\lambda a\ b. (cdcl_W-M-level-inv\ a \wedge no-strange-atm\ a)$   
     $\wedge cdcl_W-cp\ a\ b)^{**}\ T\ U$   
    **using** *cp* **by** *auto*  
    **then show** *?case* **using** *IH* **by** *auto*  
**qed**  
**qed**

**lemma** *cdcl\_W-cp-normalized-element*:  
**assumes**  
    *lev: cdcl\_W-M-level-inv*  $S$  **and**  
    *no-strange-atm*  $S$   
**obtains**  $T$  **where** *full* *cdcl\_W-cp*  $S\ T$   
**proof** –  
**let**  $?inv = \lambda a. (cdcl_W-M-level-inv\ a \wedge no-strange-atm\ a)$   
**obtain**  $T$  **where**  $T: full\ (\lambda a\ b. ?inv\ a \wedge cdcl_W-cp\ a\ b)\ S\ T$   
    **using** *cdcl\_W-cp-wf* *wf-exists-normal-form*[*of*  $\lambda a\ b. ?inv\ a \wedge cdcl_W-cp\ a\ b$ ]  
    **unfolding** *full-def* **by** *blast*  
**then have**  $cdcl_W-cp^{**}\ S\ T$   
    **using** *rtranclp-cdcl\_W-all-struct-inv-cdcl\_W-cp-iff-rtranclp-cdcl\_W-cp* *assms* **unfolding** *full-def*  
    **by** *blast*  
**moreover**  
    **then have**  $cdcl_W^{**}\ S\ T$   
    **using** *rtranclp-cdcl\_W-cp-rtranclp-cdcl\_W* **by** *blast*  
    **then have**  
    *cdcl\_W-M-level-inv*  $T$  **and**  
    *no-strange-atm*  $T$   
    **using**  $\langle cdcl_W^{**}\ S\ T \rangle$  **apply** (*simp* *add: assms(1) rtranclp-cdcl\_W-consistent-inv*)  
    **using**  $\langle cdcl_W^{**}\ S\ T \rangle$  *assms(2) rtranclp-cdcl\_W-no-strange-atm-inv* *lev* **by** *blast*  
    **then have** *no-step* *cdcl\_W-cp*  $T$   
    **using**  $T$  **unfolding** *full-def* **by** *auto*  
    **ultimately show** *thesis* **using** *that* **unfolding** *full-def* **by** *blast*  
**qed**

**lemma** *in-atms-of-implies-atm-of-on-atms-of-m*:  
 $C + \{\#L\# \} \in \# A \implies x \in atms-of\ C \implies x \in atms-of-mu\ A$   
**by** (*metis* *add.commute* *atm-iff-pos-or-neg-lit* *atms-of-atms-of-m-mono* *contra-subsetD*  
    *mem-set-mset-iff* *multi-member-skip*)

**lemma** *propagate-no-strange-atm*:  
**assumes**  
    *propagate*  $S\ S'$  **and**  
    *no-strange-atm*  $S$   
**shows** *no-strange-atm*  $S'$   
**using** *assms* **by** *induction*  
*(auto simp add: no-strange-atm-def clauses-def in-plus-implies-atm-of-on-atms-of-m*  
    *in-atms-of-implies-atm-of-on-atms-of-m)*

**lemma** *always-exists-full-cdcl<sub>W</sub>-cp-step*:

**assumes** *no-strange-atm S*

**shows**  $\exists S''. \text{full cdcl}_W\text{-cp } S \ S''$

**using** *assms*

**proof** (*induct card (atms-of-mu (init-clss S) - atm-of 'lits-of (trail S)) arbitrary: S*)

**case 0** **note** *card = this(1)* **and** *alien = this(2)*

**then have** *atm: atms-of-mu (init-clss S) = atm-of 'lits-of (trail S)*

**unfolding** *no-strange-atm-def* **by** *auto*

**{ assume** *a:  $\exists S'. \text{conflict } S \ S'$*

**then obtain** *S' where S': conflict S S' by metis*

**then have**  $\forall S''. \neg \text{cdcl}_W\text{-cp } S' \ S''$  **by** *auto*

**then have** *?case using a S' cdcl<sub>W</sub>-cp.conflict' unfolding full-def by blast*

**}**

**moreover {**

**assume** *a:  $\exists S'. \text{propagate } S \ S'$*

**then obtain** *S' where propagate S S' by blast*

**then obtain** *M N U k C L where S: state S = (M, N, U, k, C-True)*

**and** *S': state S' = (Propagated L ( (C + {#L#})) # M, N, U, k, C-True)*

**and** *C + {#L#}  $\in$  # clauses S*

**and** *M  $\models_{as}$  CNot C*

**and** *undefined-lit M L*

**using** *propagate by auto*

**have** *atms-of-mu U  $\subseteq$  atms-of-mu N* **using** *alien S unfolding no-strange-atm-def by auto*

**then have** *atm-of L  $\in$  atms-of-mu (init-clss S)*

**using**  *$\langle C + \{ \#L\# \} \in \# \text{ clauses } S \rangle$  S unfolding atms-of-m-def clauses-def by force+*

**then have** *False using  $\langle \text{undefined-lit } M \ L \rangle$  S unfolding atm unfolding lits-of-def*

**by** (*auto simp add: defined-lit-map*)

**}**

**ultimately show** *?case by (metis cdcl<sub>W</sub>-cp.cases full-def rtranclp.rtrancl-refl)*

**next**

**case** (*Suc n*) **note** *IH = this(1)* **and** *card = this(2)* **and** *alien = this(3)*

**{ assume** *a:  $\exists S'. \text{conflict } S \ S'$*

**then obtain** *S' where S': conflict S S' by metis*

**then have**  $\forall S''. \neg \text{cdcl}_W\text{-cp } S' \ S''$  **by** *auto*

**then have** *?case unfolding full-def Ex-def using S' cdcl<sub>W</sub>-cp.conflict' by blast*

**}**

**moreover {**

**assume** *a:  $\exists S'. \text{propagate } S \ S'$*

**then obtain** *S' where propagate: propagate S S' by blast*

**then obtain** *M N U k C L where*

*S: state S = (M, N, U, k, C-True) and*

*S': state S' = (Propagated L ( (C + {#L#})) # M, N, U, k, C-True) and*

*C + {#L#}  $\in$  # clauses S and*

*M  $\models_{as}$  CNot C and*

*undefined-lit M L*

**by** *fastforce*

**then have** *atm-of L  $\notin$  atm-of 'lits-of M*

**unfolding** *lits-of-def by (auto simp add: defined-lit-map)*

**moreover**

**have** *no-strange-atm S' using alien propagate propagate-no-strange-atm by blast*

**then have** *atm-of L  $\in$  atms-of-mu N* **using** *S' unfolding no-strange-atm-def by auto*

**then have**  $\bigwedge A. \{ \text{atm-of } L \} \subseteq \text{atms-of-mu } N - A \vee \text{atm-of } L \in A$  **by** *force*

**moreover have** *Suc n - card {atm-of L} = n* **by** *simp*

**moreover have** *card (atms-of-mu N - atm-of 'lits-of M) = Suc n*



```

using card S S' by simp
ultimately
  have card (atms-of-mu N - atm-of 'insert L (lits-of M)) = n
    by (metis (no-types) Diff-insert card-Diff-subset finite.emptyI finite.insertI image-insert)
  then have n = card (atms-of-mu (init-clss S') - atm-of 'lits-of (trail S'))
    using card S S' by simp
  then have a1: Ex (full cdclW-cp S') using IH ⟨no-strange-atm S'⟩ by blast
  have ?case
    proof -
      obtain S'' :: 'st where
        ff1: cdclW-cp** S' S'' ∧ no-step cdclW-cp S''
        using a1 unfolding full-def by blast
      have cdclW-cp** S S''
        using ff1 cdclW-cp.intros(2)[OF propagate]
        by (metis (no-types) converse-rtranclp-into-rtranclp)
      then have ∃ S''. cdclW-cp** S S'' ∧ (∀ S'''. ¬ cdclW-cp S'' S''')
        using ff1 by blast
      then show ?thesis unfolding full-def
        by meson
    qed
  }
ultimately show ?case unfolding full-def by (metis cdclW-cp.cases rtranclp.rtrancl-refl)
qed

```

### 17.6.3 Literal of highest level in conflicting clauses

One important property of the  $cdcl_W$  with strategy is that, whenever a conflict takes place, there is at least a literal of level  $k$  involved (except if we have derived the false clause). The reason is that we apply conflicts before a decision is taken.

**abbreviation** *no-clause-is-false* :: 'st  $\Rightarrow$  bool **where**

*no-clause-is-false*  $\equiv$

$\lambda S. (\text{conflicting } S = C\text{-True} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{as} C\text{Not } D))$

**abbreviation** *conflict-is-false-with-level* :: 'st  $\Rightarrow$  bool **where**

*conflict-is-false-with-level*  $S' \equiv \forall D. \text{conflicting } S' = C\text{-Clause } D \longrightarrow D \neq \{\#\}$   
 $\longrightarrow (\exists L \in \# D. \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S')$

**lemma** *not-conflict-not-any-negated-init-clss*:

**assumes**  $\forall S'. \neg \text{conflict } S S'$

**shows** *no-clause-is-false*  $S$

**using** *assms state-eq-ref* **by** blast

**lemma** *full-cdcl<sub>W</sub>-cp-not-any-negated-init-clss*:

**assumes** *full cdcl<sub>W</sub>-cp*  $S S'$

**shows** *no-clause-is-false*  $S'$

**using** *assms not-conflict-not-any-negated-init-clss* **unfolding** *full-def* **by** blast

**lemma** *full1-cdcl<sub>W</sub>-cp-not-any-negated-init-clss*:

**assumes** *full1 cdcl<sub>W</sub>-cp*  $S S'$

**shows** *no-clause-is-false*  $S'$

**using** *assms not-conflict-not-any-negated-init-clss* **unfolding** *full1-def* **by** blast

**lemma** *cdcl<sub>W</sub>-stgy-not-non-negated-init-clss*:

**assumes** *cdcl<sub>W</sub>-stgy*  $S S'$

**shows** *no-clause-is-false*  $S'$

```

using assms apply (induct rule: cdclW-stgy.induct)
using full1-cdclW-cp-not-any-negated-init-clss full-cdclW-cp-not-any-negated-init-clss by metis+

lemma rtrancpl-cdclW-stgy-not-non-negated-init-clss:
  assumes cdclW-stgy** S S' and no-clause-is-false S
  shows no-clause-is-false S'
  using assms by (induct rule: rtrancpl-induct) (auto simp: cdclW-stgy-not-non-negated-init-clss)

lemma cdclW-stgy-conflict-ex-lit-of-max-level:
  assumes cdclW-cp S S'
  and no-clause-is-false S
  and cdclW-M-level-inv S
  shows conflict-is-false-with-level S'
  using assms
proof (induct rule: cdclW-cp.induct)
  case conflict'
  then show ?case by auto
next
  case propagate'
  then show ?case by auto
qed

lemma no-chained-conflict:
  assumes conflict S S'
  and conflict S' S''
  shows False
  using assms by fastforce

lemma rtrancpl-cdclW-cp-propa-or-propa-conf:
  assumes cdclW-cp** S U
  shows propagate** S U  $\vee$  ( $\exists T. \text{propagate** } S T \wedge \text{conflict } T U$ )
  using assms
proof induction
  case base
  then show ?case by auto
next
  case (step U V) note SU = this(1) and UV = this(2) and IH = this(3)
  consider (confl) T where propagate** S T and conflict T U
  | (propa) propagate** S U using IH by auto
  then show ?case
  proof cases
    case confl
    then have False using UV by auto
    then show ?thesis by fast
  next
    case propa
    also have conflict U V  $\vee$  propagate U V using UV by (auto simp add: cdclW-cp.simps)
    ultimately show ?thesis by force
  qed
qed

lemma rtrancpl-cdclW-co-conflict-ex-lit-of-max-level:
  assumes full: full cdclW-cp S U
  and cls-f: no-clause-is-false S
  and conflict-is-false-with-level S

```

and  $lev: cdcl_W\text{-}M\text{-level-inv } S$   
 shows *conflict-is-false-with-level*  $U$   
**proof** (*intro allI impI*)  
 fix  $D$   
 assume *confl*: *conflicting*  $U = C\text{-Clause } D$  and  
    $D: D \neq \{\#\}$   
 consider ( $CT$ ) *conflicting*  $S = C\text{-True} \mid (SD) D'$  where *conflicting*  $S = C\text{-Clause } D'$   
   by (*cases conflicting S*) *auto*  
 then show  $\exists L \in \#D. \text{get-level } L (\text{trail } U) = \text{backtrack-lvl } U$   
**proof** *cases*  
   case  $SD$   
   then have  $S = U$   
     by (*metis (no-types) assms(1) cdcl\_W-cp-conflicting-not-empty full-def rtranclpD tranclpD*)  
   then show ?thesis using *assms(3) confl D* by *blast*—  
 next  
   case  $CT$   
   have *init-clss*  $U = \text{init-clss } S$  and *learned-clss*  $U = \text{learned-clss } S$   
     using *assms(1) unfolding full-def*  
     apply (*metis (no-types) rtranclpD tranclp-cdcl\_W-cp-no-more-init-clss*)  
     by (*metis (mono-tags, lifting) assms(1) full-def rtranclp-cdcl\_W-cp-learned-clause-inv*)  
   obtain  $T$  where *propagate\*\**  $S T$  and  $TU: \text{conflict } T U$   
   **proof** —  
     have  $f5: U \neq S$   
     using *confl CT* by *force*  
     then have  $cdcl_W\text{-}cp^{++} S U$   
       by (*metis full full-def rtranclpD*)  
     have  $\bigwedge p \text{ pa. } \neg \text{propagate } p \text{ pa} \vee \text{conflicting } pa =$   
       ( $C\text{-True}::'v \text{ literal multiset conflicting-clause}$ )  
     by *auto*  
     then show ?thesis  
       using  $f5$  that *tranclp-cdcl\_W-cp-propagate-with-conflict-or-not*[ $OF \langle cdcl_W\text{-}cp^{++} S U \rangle$ ]  
       *full confl CT unfolding full-def* by *auto*  
   qed  
   have *init-clss*  $T = \text{init-clss } S$  and *learned-clss*  $T = \text{learned-clss } S$   
     using  $TU \langle \text{init-clss } U = \text{init-clss } S \rangle \langle \text{learned-clss } U = \text{learned-clss } S \rangle$  by *auto*  
   then have  $D \in \# \text{clauses } S$   
     using  $TU \text{ confl}$  by (*fastforce simp: clauses-def*)  
   then have  $\neg \text{trail } S \models_{as} C\text{Not } D$   
     using *cls-f CT* by *simp*  
   moreover  
     obtain  $M$  where *tr-U*:  $\text{trail } U = M @ \text{trail } S$  and *nm*:  $\forall m \in \text{set } M. \neg \text{is-marked } m$   
       by (*metis (mono-tags, lifting) assms(1) full-def rtranclp-cdcl\_W-cp-dropWhile-trail*)  
     have  $\text{trail } U \models_{as} C\text{Not } D$   
       using  $TU \text{ confl}$  by *auto*  
   ultimately obtain  $L$  where  $L \in \# D$  and  $\neg L \in \text{lits-of } M$   
     *unfolding tr-U CNot-def true-annots-def Ball-def true-annot-def true-cls-def* by *auto*  
  
 moreover have *inv-U*:  $cdcl_W\text{-}M\text{-level-inv } U$   
   by (*metis cdcl\_W-stgy.conflict' cdcl\_W-stgy-consistent-inv full full-unfold lev*)  
 moreover  
   have  $\text{backtrack-lvl } U = \text{backtrack-lvl } S$   
     using *full unfolding full-def* by (*auto dest: rtranclp-cdcl\_W-cp-backtrack-lvl*)  
  
 moreover  
   have *no-dup* ( $\text{trail } U$ )

```

    using inv-U unfolding cdclW-M-level-inv-def by auto
  { fix x :: ('v, nat, 'v literal multiset) marked-lit and
    xb :: ('v, nat, 'v literal multiset) marked-lit
    assume a1: atm-of L = atm-of (lit-of xb)
    moreover assume a2: - L = lit-of x
    moreover assume a3: (λl. atm-of (lit-of l)) ' set M
      ∩ (λl. atm-of (lit-of l)) ' set (trail S) = {}
    moreover assume a4: x ∈ set M
    moreover assume a5: xb ∈ set (trail S)
    moreover have atm-of (- L) = atm-of L
      by auto
    ultimately have False
      by auto
  }
  then have LS: atm-of L ∉ atm-of ' lits-of (trail S)
    using ⟨-L ∈ lits-of M⟩ ⟨no-dup (trail U)⟩ unfolding tr-U lits-of-def by auto
  ultimately have get-level L (trail U) = backtrack-lvl U
  proof (cases get-all-levels-of-marked (trail S) ≠ [], goal-cases)
    case 2 note LD = this(1) and LM = this(2) and inv-U = this(3) and US = this(4) and
      LS = this(5) and ne = this(6)
    have backtrack-lvl S = 0
      using lev ne unfolding cdclW-M-level-inv-def by auto
    moreover have get-rev-level L 0 (rev M) = 0
      using nm by auto
    ultimately show ?thesis using LS ne US unfolding tr-U
      by (simp add: get-all-levels-of-marked-nil-iff-not-is-marked lits-of-def)
  next
    case 1 note LD = this(1) and LM = this(2) and inv-U = this(3) and US = this(4) and
      LS = this(5) and ne = this(6)

    have hd (get-all-levels-of-marked (trail S)) = backtrack-lvl S
      using ne lev unfolding cdclW-M-level-inv-def
      by (cases get-all-levels-of-marked (trail S)) auto
    moreover have atm-of L ∈ atm-of ' lits-of M
      using ⟨-L ∈ lits-of M⟩ by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
        lits-of-def)
    ultimately show ?thesis
      using nm ne unfolding tr-U
      using get-level-skip-beginning-hd-get-all-levels-of-marked[OF LS, of M]
        get-level-skip-in-all-not-marked[of rev M L backtrack-lvl S]
      unfolding lits-of-def US
      by auto
    qed
  then show ∃ L ∈ #D. get-level L (trail U) = backtrack-lvl U
    using ⟨L ∈ # D⟩ by blast
  qed
qed

```

#### 17.6.4 Literal of highest level in marked literals

**definition** *mark-is-false-with-level* :: 'st ⇒ bool **where**

*mark-is-false-with-level* S' ≡

$\forall D \ M1 \ M2 \ L. \ M1 \ @ \ \text{Propagated } L \ D \ \# \ M2 = \text{trail } S' \longrightarrow D - \{\#L\} \neq \{\#\}$   
 $\longrightarrow (\exists L. L \in \# \ D \wedge \text{get-level } L \ (\text{trail } S') = \text{get-maximum-possible-level } M1)$

**definition** *no-more-propagation-to-do* :: 'st ⇒ bool **where**

$no-more-propagation-to-do\ S \equiv$   
 $\forall D\ M\ M'\ L.\ D + \{\#L\#\} \in \# \text{ clauses } S \longrightarrow \text{trail } S = M' @ M \longrightarrow M \models_{as} CNot\ D$   
 $\longrightarrow \text{undefined-lit } M\ L \longrightarrow \text{get-maximum-possible-level } M < \text{backtrack-lvl } S$   
 $\longrightarrow (\exists L.\ L \in \# D \wedge \text{get-level } L (\text{trail } S) = \text{get-maximum-possible-level } M)$

**lemma** *propagate-no-more-propagation-to-do*:

**assumes** *propagate*: *propagate*  $S\ S'$   
**and**  $H$ : *no-more-propagation-to-do*  $S$   
**and**  $M$ : *cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows** *no-more-propagation-to-do*  $S'$   
**using** *assms*

**proof** –

**obtain**  $M\ N\ U\ k\ C\ L$  **where**

$S$ : state  $S = (M, N, U, k, C-True)$  **and**

$S'$ : state  $S' = (Propagated\ L\ ( (C + \{\#L\#\})) \# M, N, U, k, C-True)$  **and**

$C + \{\#L\#\} \in \# \text{ clauses } S$  **and**

$M \models_{as} CNot\ C$  **and**

*undefined-lit*  $M\ L$

**using** *propagate* **by** *auto*

**let**  $?M' = Propagated\ L\ ( (C + \{\#L\#\})) \# M$

**show** *?thesis unfolding no-more-propagation-to-do-def*

**proof** (*intro allI impI*)

**fix**  $D\ M1\ M2\ L'$

**assume**  $D-L$ :  $D + \{\#L'\#\} \in \# \text{ clauses } S'$

**and**  $\text{trail } S' = M2 @ M1$

**and** *get-max*:  $\text{get-maximum-possible-level } M1 < \text{backtrack-lvl } S'$

**and**  $M1 \models_{as} CNot\ D$

**and** *undef*: *undefined-lit*  $M1\ L'$

**have**  $tl\ M2 @ M1 = \text{trail } S \vee (M2 = [] \wedge M1 = Propagated\ L\ ( (C + \{\#L\#\})) \# M)$

**using**  $\langle \text{trail } S' = M2 @ M1 \rangle S'\ S$  **by** (*cases*  $M2$ ) *auto*

**moreover** {

**assume**  $tl\ M2 @ M1 = \text{trail } S$

**moreover** **have**  $D + \{\#L'\#\} \in \# \text{ clauses } S$  **using**  $D-L\ S\ S'$  *unfolding clauses-def* **by** *auto*

**moreover** **have**  $\text{get-maximum-possible-level } M1 < \text{backtrack-lvl } S$

**using** *get-max*  $S\ S'$  **by** *auto*

**ultimately** **obtain**  $L'$  **where**  $L' \in \# D$  **and**

$\text{get-level } L' (\text{trail } S) = \text{get-maximum-possible-level } M1$

**using**  $H\ \langle M1 \models_{as} CNot\ D \rangle \text{undef}$  *unfolding no-more-propagation-to-do-def* **by** *metis*

**moreover**

{ **have** *cdcl<sub>W</sub>-M-level-inv*  $S'$

**using** *cdcl<sub>W</sub>-consistent-inv*[ $OF - M$ ] *cdcl<sub>W</sub>.propagate*[ $OF\ propagate$ ] **by** *blast*

**then** **have** *no-dup*  $?M'$  **using**  $S'$  *unfolding cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*

**moreover**

**have** *atm-of*  $L' \in \text{atm-of } (lits\ of\ M1)$

**using**  $\langle L' \in \# D \rangle \langle M1 \models_{as} CNot\ D \rangle$  **by** (*metis atm-of-uminus image-eqI in-CNot-implies-uminus*(2))

**then** **have** *atm-of*  $L' \in \text{atm-of } (lits\ of\ M)$

**using**  $\langle tl\ M2 @ M1 = \text{trail } S \rangle S$  **by** *auto*

**ultimately** **have**  $\text{atm-of } L \neq \text{atm-of } L'$  *unfolding lits-of-def* **by** *auto*

}

**ultimately** **have**  $\exists L' \in \# D.\ \text{get-level } L' (\text{trail } S') = \text{get-maximum-possible-level } M1$

**using**  $S\ S'$  **by** *auto*

}

**moreover** {

**assume**  $M2 = []$  **and**  $M1$ :  $M1 = Propagated\ L\ ( (C + \{\#L\#\})) \# M$

```

    have cdclW-M-level-inv S'
      using cdclW-consistent-inv[OF - M] cdclW.propagate[OF propagate] by blast
    then have get-all-levels-of-marked (trail S') = rev ([Suc 0..<(Suc 0+k)])
      using S' unfolding cdclW-M-level-inv-def by auto
    then have get-maximum-possible-level M1 = backtrack-lvl S'
      using get-maximum-possible-level-max-get-all-levels-of-marked[of M1] S' M1
      by (auto intro: Max-eqI)
    then have False using get-max by auto
  }
  ultimately show  $\exists L. L \in \# D \wedge \text{get-level } L \text{ (trail } S') = \text{get-maximum-possible-level } M1$  by fast
qed
qed

```

```

lemma conflict-no-more-propagation-to-do:
  assumes conflict: conflict S S'
  and H: no-more-propagation-to-do S
  and M: cdclW-M-level-inv S
  shows no-more-propagation-to-do S'
  using assms unfolding no-more-propagation-to-do-def conflict.simps by force

```

```

lemma cdclW-cp-no-more-propagation-to-do:
  assumes conflict: cdclW-cp S S'
  and H: no-more-propagation-to-do S
  and M: cdclW-M-level-inv S
  shows no-more-propagation-to-do S'
  using assms
  proof (induct rule: cdclW-cp.induct)
  case (conflict' S S')
  then show ?case using conflict-no-more-propagation-to-do[of S S'] by blast
next
  case (propagate' S S') note S = this
  show 1: no-more-propagation-to-do S'
    using propagate-no-more-propagation-to-do[of S S'] S by blast
qed

```

```

lemma cdclW-then-exists-cdclW-stgy-step:
  assumes
    o: cdclW-o S S' and
    alien: no-strange-atm S and
    lev: cdclW-M-level-inv S
  shows  $\exists S'. \text{cdcl}_W\text{-stgy } S S'$ 
proof –
  obtain S'' where full cdclW-cp S' S''
    using always-exists-full-cdclW-cp-step alien cdclW-no-strange-atm-inv cdclW-o-no-more-init-clss
    o other lev by (meson cdclW-consistent-inv)
  then show ?thesis
    using assms by (metis always-exists-full-cdclW-cp-step cdclW-stgy.conflict' full-unfold other')
qed

```

```

lemma backtrack-no-decomp:
  assumes S: state S = (M, N, U, k, C-Clause (D + {#L#}))
  and L: get-level L M = k
  and D: get-maximum-level D M < k
  and M-L: cdclW-M-level-inv S
  shows  $\exists S'. \text{cdcl}_W\text{-o } S S'$ 

```

```

proof –
  have  $L-D$ :  $\text{get-level } L \ M = \text{get-maximum-level } (D + \{\#L\}) \ M$ 
    using  $L \ D$  by ( $\text{simp add: get-maximum-level-plus}$ )
  let  $?i = \text{get-maximum-level } D \ M$ 
  obtain  $K \ M1 \ M2$  where  $K$ :  $(\text{Marked } K \ (?i + 1) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
    using  $\text{backtrack-ex-decomp}[OF \ M-L, \text{ of } ?i] \ D \ S$  by  $\text{auto}$ 
  show  $?thesis$  using  $\text{backtrack-rule}[OF \ S \ K \ L \ L-D]$  by ( $\text{meson } bj \ \text{cdcl}_W\text{-bj.simps state-eq-ref}$ )
qed

```

```

lemma  $\text{cdcl}_W\text{-stgy-final-state-conclusive}$ :
  assumes  $\text{termi}$ :  $\forall S'. \neg \text{cdcl}_W\text{-stgy } S \ S'$ 
  and  $\text{decomp}$ :  $\text{all-decomposition-implies-m } (\text{init-clss } S) \ (\text{get-all-marked-decomposition } (\text{trail } S))$ 
  and  $\text{learned}$ :  $\text{cdcl}_W\text{-learned-clause } S$ 
  and  $\text{level-inv}$ :  $\text{cdcl}_W\text{-M-level-inv } S$ 
  and  $\text{alien}$ :  $\text{no-strange-atm } S$ 
  and  $\text{no-dup}$ :  $\text{distinct-cdcl}_W\text{-state } S$ 
  and  $\text{confl}$ :  $\text{cdcl}_W\text{-conflicting } S$ 
  and  $\text{confl-k}$ :  $\text{conflict-is-false-with-level } S$ 
  shows  $(\text{conflicting } S = C\text{-Clause } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S)))$ 
     $\vee (\text{conflicting } S = C\text{-True} \wedge \text{trail } S \models_{\text{as}} \text{set-mset } (\text{init-clss } S))$ 

```

```

proof –
  let  $?M = \text{trail } S$ 
  let  $?N = \text{init-clss } S$ 
  let  $?k = \text{backtrack-lvl } S$ 
  let  $?U = \text{learned-clss } S$ 
  have  $\text{conflicting } S = C\text{-Clause } \{\#\}$ 
     $\vee \text{conflicting } S = C\text{-True}$ 
     $\vee (\exists D \ L. \text{conflicting } S = C\text{-Clause } (D + \{\#L\}))$ 
  apply ( $\text{case-tac } \text{conflicting } S, \text{ auto}$ )
  by ( $\text{case-tac } x2, \text{ auto}$ )
moreover {
  assume  $\text{conflicting } S = C\text{-Clause } \{\#\}$ 
  then have  $\text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$ 
    using  $\text{assms}(3)$  unfolding  $\text{cdcl}_W\text{-learned-clause-def true-clss-cls-def}$ 
    by ( $\text{metis } (\text{no-types}, \text{lifting}) \text{Un-insert-right atms-of-empty satisfiable-def}$ 
       $\text{sup-bot.right-neutral total-over-m-insert total-over-set-empty true-clss-empty}$ )
  }
moreover {
  assume  $\text{conflicting } S = C\text{-True}$ 
  { assume  $\neg ?M \models_{\text{asm}} ?N$ 
    have  $\text{atm-of } ' \ (\text{lits-of } ?M) = \text{atms-of-mu } ?N \ (\text{is } ?A = ?B)$ 
    proof
      show  $?A \subseteq ?B$  using  $\text{alien unfolding no-strange-atm-def}$  by  $\text{auto}$ 
      show  $?B \subseteq ?A$ 
      proof ( $\text{rule ccontr}$ )
        assume  $\neg ?B \subseteq ?A$ 
        then obtain  $l$  where  $l \in ?B$  and  $l \notin ?A$  by  $\text{auto}$ 
        then have  $\text{undefined-lit } ?M \ (\text{Pos } l)$ 
          using  $\langle l \notin ?A \rangle$  unfolding  $\text{lits-of-def}$  by ( $\text{auto simp add: defined-lit-map}$ )
        then have  $\exists S'. \text{cdcl}_W\text{-o } S \ S'$ 
          using  $\text{cdcl}_W\text{-o.decide decide.intros } \langle l \in ?B \rangle \text{ no-strange-atm-def}$ 
          by ( $\text{metis } \langle \text{conflicting } S = C\text{-True} \rangle \text{literal.sel}(1) \text{state-eq-def}$ )
        then show  $\text{False}$ 
          using  $\text{termi cdcl}_W\text{-then-exists-cdcl}_W\text{-stgy-step}[OF \text{ - alien}] \text{ level-inv}$  by  $\text{blast}$ 
      end
    end
  }

```

```

    qed
  qed
  obtain  $D$  where  $\neg ?M \models_a D$  and  $D \in \# ?N$ 
    using  $\langle \neg ?M \models_{asm} ?N \rangle$  unfolding lits-of-def true-annots-def Ball-def by auto
  have  $atms\text{-}of\ D \subseteq atm\text{-}of\ \langle lits\text{-}of\ ?M \rangle$ 
    using  $\langle D \in \# ?N \rangle$  unfolding  $\langle atm\text{-}of\ \langle lits\text{-}of\ ?M \rangle = atms\text{-}of\text{-}mu\ ?N \rangle$  atms-of-m-def
    by (auto simp add: atms-of-def)
  then have  $a1: atm\text{-}of\ \langle set\text{-}mset\ D \subseteq atm\text{-}of\ \langle lits\text{-}of\ (trail\ S) \rangle$ 
    by (auto simp add: atms-of-def lits-of-def)
  have  $total\text{-}over\text{-}m\ (lits\text{-}of\ ?M)\ \{D\}$ 
    using  $\langle atms\text{-}of\ D \subseteq atm\text{-}of\ \langle lits\text{-}of\ ?M \rangle \rangle$  atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
    by (fastforce simp: total-over-set-def)
  then have  $?M \models_{as}\ CNot\ D$ 
    using  $total\text{-}not\text{-}true\text{-}cls\text{-}true\text{-}clss\text{-}CNot\ \langle \neg trail\ S \models_a D \rangle$  true-annot-def
    true-annots-true-cls by fastforce
  then have False
  proof -
    obtain  $S'$  where
       $f2: full\ cdcl_W\text{-}cp\ S\ S'$ 
      by (meson alien always-exists-full-cdcl_W-cp-step level-inv)
    then have  $S' = S$ 
      using  $cdcl_W\text{-}stgy.conflict'[of\ S]$  by (metis (no-types) full-unfold termi)
    then show  $?thesis$ 
      using  $f2\ \langle D \in \# init\text{-}clss\ S \rangle\ \langle conflicting\ S = C\text{-}True \rangle\ \langle trail\ S \models_{as}\ CNot\ D \rangle$ 
      clauses-def full-cdcl_W-cp-not-any-negated-init-clss by auto
  qed
}
then have  $?M \models_{asm}\ ?N$  by blast
}
moreover {
  assume  $\exists D\ L.\ conflicting\ S = C\text{-}Clause\ (D + \{\#L\# \})$ 
  obtain  $D\ L$  where  $LD: conflicting\ S = C\text{-}Clause\ (D + \{\#L\# \})$  and  $get\text{-}level\ L\ ?M = ?k$ 
  proof -
    obtain  $mm :: 'v\ literal\ multiset$  and  $ll :: 'v\ literal$  where
       $f2: conflicting\ S = C\text{-}Clause\ (mm + \{\#ll\# \})$ 
      using  $\langle \exists D\ L.\ conflicting\ S = C\text{-}Clause\ (D + \{\#L\# \}) \rangle$  by force
    have  $\forall m.\ (conflicting\ S \neq C\text{-}Clause\ m \vee m = \{\# \})$ 
       $\vee (\exists l.\ l \in \# m \wedge get\text{-}level\ l\ (trail\ S) = backtrack\text{-}lvl\ S)$ 
      using  $confl\text{-}k$  by blast
    then show  $?thesis$ 
      using  $f2$  that by (metis (no-types) multi-member-split single-not-empty union-eq-empty)
  qed
  let  $?D = D + \{\#L\# \}$ 
  have  $?D \neq \{\# \}$  by auto
  have  $?M \models_{as}\ CNot\ ?D$  using  $confl\ LD$  unfolding  $cdcl_W\text{-}conflicting\text{-}def$  by auto
  then have  $?M \neq []$  unfolding true-annots-def Ball-def true-annot-def true-cls-def by force
  { have  $M: ?M = hd\ ?M\ \# tl\ ?M$  using  $\langle ?M \neq [] \rangle$  list.collapse by fastforce
    assume  $marked: is\text{-}marked\ (hd\ ?M)$ 
    then obtain  $k'$  where  $k': k' + 1 = ?k$ 
      using  $level\text{-}inv\ M$  unfolding  $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ 
      by (cases  $hd\ (trail\ S)$ ; cases  $trail\ S$ ) auto
    obtain  $L'\ l'$  where  $L': hd\ ?M = Marked\ L'\ l'$  using  $marked$  by (case-tac  $hd\ ?M$ ) auto
    have  $get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (hd\ (trail\ S)\ \# tl\ (trail\ S))$ 
       $= rev\ [1..<1 + length\ (get\text{-}all\text{-}levels\text{-}of\text{-}marked\ ?M)]$ 
    using  $level\text{-}inv\ \langle get\text{-}level\ L\ ?M = ?k \rangle\ M$  unfolding  $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def\ M[symmetric]$ 
  }
}

```



by *blast*  
 then have  $l'-tl: l' \# \text{get-all-levels-of-marked } (tl \ ?M)$   
   =  $\text{rev } [1..<1 + \text{length } (\text{get-all-levels-of-marked } ?M)]$  **unfolding**  $L'$  by *simp*  
 moreover have  $\dots = \text{length } (\text{get-all-levels-of-marked } ?M)$   
   #  $\text{rev } [1..<\text{length } (\text{get-all-levels-of-marked } ?M)]$   
   **using**  $M \text{ Suc-le-mono calculation by } (\text{fastforce simp add: upt.simps}(2))$   
 finally have  
    $l' = ?k$  **and**  
    $g-r: \text{get-all-levels-of-marked } (tl \ (\text{trail } S))$   
     =  $\text{rev } [1..<\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$   
   **using**  $\text{level-inv } \langle \text{get-level } L \ ?M = ?k \rangle M$  **unfolding**  $\text{cdcl}_W\text{-M-level-inv-def}$  by *auto*  
 have \*:  $\bigwedge \text{list. no-dup list} \implies$   
    $-L \in \text{lits-of list} \implies \text{atm-of } L \in \text{atm-of ' lits-of list}$   
   by  $(\text{metis atm-of-uminus imageI})$   
 have  $L' = -L$   
**proof** (rule *ccontr*)  
   assume  $\neg ?thesis$   
   moreover have  $-L \in \text{lits-of } ?M$  **using**  $\text{confl } LD$  **unfolding**  $\text{cdcl}_W\text{-conflicting-def}$  by *auto*  
   ultimately have  $\text{get-level } L \ (\text{hd } (\text{trail } S) \# \text{tl } (\text{trail } S)) = \text{get-level } L \ (tl \ ?M)$   
     **using**  $\text{cdcl}_W\text{-M-level-inv-decomp}(1)[OF \ \text{level-inv}]$  **unfolding**  $L'$  *consistent-interp-def*  
     by  $(\text{metis } (\text{no-types, lifting}) \ L' \ M \ \text{atm-of-eq-atm-of get-level-skip-beginning insert-iff}$   
        $\text{lits-of-cons marked-lit.sel}(1))$   
  
   moreover  
   have  $\text{length } (\text{get-all-levels-of-marked } (\text{trail } S)) = ?k$   
     **using**  $\text{level-inv}$  **unfolding**  $\text{cdcl}_W\text{-M-level-inv-def}$  by *auto*  
   then have  $\text{Max } (\text{set } (0 \# \text{get-all-levels-of-marked } (tl \ (\text{trail } S)))) = ?k - 1$   
     **unfolding**  $g-r$  by  $(\text{auto simp add: Max-n-upt})$   
   then have  $\text{get-level } L \ (tl \ ?M) < ?k$   
     **using**  $\text{get-maximum-possible-level-ge-get-level}[of \ L \ tl \ ?M]$   
     by  $(\text{metis One-nat-def add.right-neutral add-Suc-right diff-add-inverse2}$   
        $\text{get-maximum-possible-level-max-get-all-levels-of-marked } k' \ \text{le-imp-less-Suc}$   
        $\text{list.simps}(15))$   
   finally show  $\text{False}$  **using**  $\langle \text{get-level } L \ ?M = ?k \rangle M$  by *auto*  
 qed  
 have  $L: \text{hd } ?M = \text{Marked } (-L) \ ?k$  **using**  $\langle l' = ?k \rangle \langle L' = -L \rangle L'$  by *auto*  
  
 have  $g-a-l: \text{get-all-levels-of-marked } ?M = \text{rev } [1..<1 + ?k]$   
   **using**  $\text{level-inv } \langle \text{get-level } L \ ?M = ?k \rangle M$  **unfolding**  $\text{cdcl}_W\text{-M-level-inv-def}$  by *auto*  
 have  $g-k: \text{get-maximum-level } D \ (\text{trail } S) \leq ?k$   
   **using**  $\text{get-maximum-possible-level-ge-get-maximum-level}[of \ D \ ?M]$   
    $\text{get-maximum-possible-level-max-get-all-levels-of-marked}[of \ ?M]$   
   by  $(\text{auto simp add: Max-n-upt } g-a-l)$   
 have  $\text{get-maximum-level } D \ (\text{trail } S) < ?k$   
**proof** (rule *ccontr*)  
   assume  $\neg ?thesis$   
   then have  $\text{get-maximum-level } D \ (\text{trail } S) = ?k$  **using**  $M \ g-k$  **unfolding**  $L$  by *auto*  
   then obtain  $L'$  where  $L' \in \# D$  **and**  $L-k: \text{get-level } L' \ ?M = ?k$   
     **using**  $\text{get-maximum-level-exists-lit}[of \ ?k \ D \ ?M]$  **unfolding**  $k'[\text{symmetric}]$  by *auto*  
   have  $L \neq L'$  **using**  $\text{no-dup } \langle L' \in \# D \rangle$   
   **unfolding**  $\text{distinct-cdcl}_W\text{-state-def } LD$  by  $(\text{metis add commute add-eq-self-zero}$   
      $\text{count-single count-union less-not-refl3 distinct-mset-def union-single-eq-member})$   
   have  $L' = -L$   
   **proof** (rule *ccontr*)  
     assume  $\neg ?thesis$

```

    then have get-level L' ?M = get-level L' (tl ?M)
      using M ⟨L ≠ L'⟩ get-level-skip-beginning[of L' hd ?M tl ?M] unfolding L
      by (auto simp add: atm-of-eq-atm-of)
    moreover have ... < ?k
      using level-inv g-r get-rev-level-less-max-get-all-levels-of-marked[of L' 0
        rev (tl ?M)] L-k l'-tl calculation g-a-l
      by (auto simp add: Max-n-upt cdclW-M-level-inv-def)
    finally show False using L-k by simp
  qed
then have taut: tautology (D + {#L#})
  using ⟨L' ∈ # D⟩ by (metis add.commute mset-leD mset-le-add-left multi-member-this
    tautology-minus)
have consistent-interp (lits-of ?M)
  using level-inv unfolding cdclW-M-level-inv-def by auto
then have ¬?M ⊨as CNot ?D
  using taut by (metis (no-types) ⟨L' = - L⟩ ⟨L' ∈ # D⟩ add.commute consistent-interp-def
    in-CNot-implies-uminus(2) mset-leD mset-le-add-left multi-member-this)
moreover have ?M ⊨as CNot ?D
  using confl no-dup LD unfolding cdclW-conflicting-def by auto
ultimately show False by blast
qed
then have False
  using backtrack-no-decomp[OF - ⟨get-level L (trail S) = backtrack-lvl S⟩ - level-inv]
  LD alien termi by (metis cdclW-then-exists-cdclW-stgy-step level-inv)
}
moreover {
  assume ¬is-marked (hd ?M)
  then obtain L' C where L'C: hd ?M = Propagated L' C by (case-tac hd ?M, auto)
  then have M: ?M = Propagated L' C # tl ?M using ⟨?M ≠ []⟩ list.collapse by fastforce
  then obtain C' where C': C = C' + {#L'#}
    using confl unfolding cdclW-conflicting-def by (metis append-Nil diff-single-eq-union)
  { assume -L' ∉ # ?D
    then have False
      using bj[OF cdclW-bj.skip[OF skip-rule[OF - ⟨-L' ∉ # ?D⟩ ⟨?D ≠ {#}⟩, of S C tl (trail S) -
        ]]]
      termi M by (metis LD alien cdclW-then-exists-cdclW-stgy-step state-eq-def level-inv)
    }
}
moreover {
  assume -L' ∈ # ?D
  then obtain D' where D': ?D = D' + {#-L'#} by (metis insert-DiffM2)
  have g-r: get-all-levels-of-marked (Propagated L' C # tl (trail S))
    = rev [Suc 0..Suc (length (get-all-levels-of-marked (trail S)))]
    using level-inv M unfolding cdclW-M-level-inv-def by auto
  have Max (insert 0 (set (get-all-levels-of-marked (Propagated L' C # tl (trail S))))) = ?k
    using level-inv M unfolding g-r cdclW-M-level-inv-def set-rev
    by (auto simp add: Max-n-upt)
  then have get-maximum-level D' (Propagated L' C # tl ?M) ≤ ?k
    using get-maximum-possible-level-ge-get-maximum-level[of D' Propagated L' C # tl ?M]
    unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
  then have get-maximum-level D' (Propagated L' C # tl ?M) = ?k
    ∨ get-maximum-level D' (Propagated L' C # tl ?M) < ?k
    using le-neq-implies-less by blast
  moreover {
    assume g-D'-k: get-maximum-level D' (Propagated L' C # tl ?M) = ?k
    have False

```

```

proof –
  have f1: get-maximum-level  $D'$  (trail  $S$ ) = backtrack-lvl  $S$ 
    using  $M$   $g\text{-}D'\text{-}k$  by auto
  have (trail  $S$ , init-clss  $S$ , learned-clss  $S$ , backtrack-lvl  $S$ , C-Clause ( $D + \{\#L\# \}$ ))
    = state  $S$ 
    by (metis (no-types)  $LD$ )
  then have  $cdcl_W\text{-}o$   $S$  (update-conflicting (C-Clause ( $D' \# \cup C'$ )) (tl-trail  $S$ ))
    using f1  $bj[OF\ cdcl_W\text{-}bj.resolve[OF\ resolve\text{-}rule[of\ S\ L'\ C'\ tl\ ?M\ ?N\ ?U\ ?k\ D']]]$ 
     $C'\ D'\ M$  by (metis state-eq-def)
  then show ?thesis
    by (meson alien  $cdcl_W\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step$  termi level-inv)
qed
}
moreover {
  assume get-maximum-level  $D'$  (Propagated  $L'\ C\ \# \text{tl}\ ?M$ ) <  $?k$ 
  then have False
    proof –
      assume a1: get-maximum-level  $D'$  (Propagated  $L'\ C\ \# \text{tl}\ (\text{trail } S)$ ) < backtrack-lvl  $S$ 
      obtain  $mm :: 'v\ literal\ multiset$  and  $ll :: 'v\ literal$  where
        f2: conflicting  $S = C\text{-Clause}\ (mm + \{\#ll\# \})$ 
        get-level  $ll\ (\text{trail } S) = \text{backtrack-lvl } S$ 
        using  $LD\ \langle \text{get-level } L\ (\text{trail } S) = \text{backtrack-lvl } S \rangle$  by blast
      then have f3: get-maximum-level  $D'$  (trail  $S$ ) ≤ get-level  $ll\ (\text{trail } S)$ 
        using  $M$  a1 by force
      have get-level  $ll\ (\text{trail } S) \neq \text{get-maximum-level } D'\ (\text{trail } S)$ 
        using f2  $M$  calculation(2) by presburger
      have f1: trail  $S = \text{Propagated } L'\ C\ \# \text{tl}\ (\text{trail } S)$ 
        conflicting  $S = C\text{-Clause}\ (D' + \{\#-\ L'\#\})$ 
        using  $D'\ LD\ M$  by force+
      have f2: conflicting  $S = C\text{-Clause}\ (mm + \{\#ll\# \})$ 
        get-level  $ll\ (\text{trail } S) = \text{backtrack-lvl } S$ 
        using f2 by force+
      have  $ll = -\ L'$ 
        by (metis (no-types)  $D'\ LD\ \langle \text{get-level } ll\ (\text{trail } S) \neq \text{get-maximum-level } D'\ (\text{trail } S) \rangle$ 
          conflicting-clause.inject f2 f3 get-maximum-level-ge-get-level insert-noteq-member
          le-antisym)
      then show ?thesis
        using f2 f1  $M$  backtrack-no-decomp[of  $S$ ]
        by (metis (no-types) a1 alien  $cdcl_W\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step$  level-inv termi)
    qed
}
ultimately have False by blast
}
ultimately have False by blast
}
ultimately have False by blast
}
ultimately show ?thesis by blast
qed

lemma  $cdcl_W\text{-}cp\text{-}tranclp\text{-}cdcl_W$ :
   $cdcl_W\text{-}cp\ S\ S' \implies cdcl_W^{++}\ S\ S'$ 
apply (induct rule: cdcl_W-cp.induct)
by (meson  $cdcl_W.conflict\ cdcl_W.propagate\ tranclp.r\text{-}into\text{-}trancl\ tranclp.trancl\text{-}into\text{-}trancl$ ) +

```

```

lemma trancpl-cdclW-cp-trancpl-cdclW:
  cdclW-cp++ S S'  $\implies$  cdclW++ S S'
  apply (induct rule: trancpl.induct)
  apply (simp add: cdclW-cp-trancpl-cdclW)
  by (meson cdclW-cp-trancpl-cdclW trancpl-trans)

lemma cdclW-stgy-trancpl-cdclW:
  cdclW-stgy S S'  $\implies$  cdclW++ S S'
proof (induct rule: cdclW-stgy.induct)
  case conflict'
  then show ?case
    unfolding full1-def by (simp add: trancpl-cdclW-cp-trancpl-cdclW)
next
  case (other' S' S'')
  then have S' = S''  $\vee$  cdclW-cp++ S' S''
    by (simp add: rtrancpl-unfold full-def)
  then show ?case
    using other' by (meson cdclW-ops.other cdclW-ops-axioms trancpl.r-into-trancpl
      trancpl-cdclW-cp-trancpl-cdclW trancpl-trans)
qed

lemma trancpl-cdclW-stgy-trancpl-cdclW:
  cdclW-stgy++ S S'  $\implies$  cdclW++ S S'
  apply (induct rule: trancpl.induct)
  using cdclW-stgy-trancpl-cdclW apply blast
  by (meson cdclW-stgy-trancpl-cdclW trancpl-trans)

lemma rtrancpl-cdclW-stgy-rtrancpl-cdclW:
  cdclW-stgy** S S'  $\implies$  cdclW** S S'
  using rtrancpl-unfold[of cdclW-stgy S S'] trancpl-cdclW-stgy-trancpl-cdclW[of S S'] by auto

lemma cdclW-o-conflict-is-false-with-level-inv:
  assumes
    cdclW-o S S' and
    lev: cdclW-M-level-inv S and
    confl-inv: conflict-is-false-with-level S and
    n-d: distinct-cdclW-state S and
    conflicting: cdclW-conflicting S
  shows conflict-is-false-with-level S'
  using assms(1,2)
proof (induct rule: cdclW-o-induct-lev2)
  case (resolve L C M D T) note tr-S = this(1) and confl = this(2) and T = this(4)
  have -L  $\notin$  D using n-d confl unfolding distinct-cdclW-state-def distinct-mset-def by auto
  moreover have L  $\notin$  D
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    moreover have Propagated L (C + {#L#})  $\#$  M  $\models_{as}$  CNot D
      using conflicting confl tr-S unfolding cdclW-conflicting-def by auto
    ultimately have -L  $\in$  lits-of (Propagated L (C + {#L#}))  $\#$  M
      using in-CNot-implies-uminus(2) by blast
    moreover have no-dup (Propagated L (C + {#L#}))  $\#$  M
      using lev tr-S unfolding cdclW-M-level-inv-def by auto
    ultimately show False unfolding lits-of-def by (metis consistent-interp-def image-eqI
      list.set-intros(1) lits-of-def marked-lit.sel(2) distinctconsistent-interp)
  qed

```

```

ultimately
  have  $g\text{-}D$ :  $\text{get-maximum-level } D \text{ (Propagated } L \text{ ( } (C + \{\#L\#\}) \text{)) } \# M$ 
    =  $\text{get-maximum-level } D M$ 
  proof -
    have  $\forall a f L. ((a::'v) \in f \text{ ' } L) = (\exists l. (l::'v \text{ literal}) \in L \wedge a = f l)$ 
      by blast
    then show ?thesis
      using  $\text{get-maximum-level-skip-first[of } L D \text{ ( } (C + \{\#L\#\}) M \text{] unfolding atm-s-of-def}$ 
      by (metis (no-types)  $\langle - L \notin \# D \rangle \langle L \notin \# D \rangle \text{atm-of-eq-atm-of mem-set-mset-iff}$ )
  qed
{ assume
   $\text{get-maximum-level } D \text{ (Propagated } L \text{ ( } (C + \{\#L\#\}) \text{)) } \# M = \text{backtrack-lvl } S$  and
   $\text{backtrack-lvl } S > 0$ 
  then have  $D$ :  $\text{get-maximum-level } D M = \text{backtrack-lvl } S$  unfolding  $g\text{-}D$  by blast
  then have ?case
    using  $\text{tr-}S \langle \text{backtrack-lvl } S > 0 \rangle \text{get-maximum-level-exists-lit[of backtrack-lvl } S D M] T$ 
    by auto
}
moreover {
  assume [simp]:  $\text{backtrack-lvl } S = 0$ 
  have  $\bigwedge L. \text{get-level } L M = 0$ 
  proof -
    fix  $L$ 
    have  $\text{atm-of } L \notin \text{atm-of ' (lits-of } M) \implies \text{get-level } L M = 0$  by auto
    moreover {
      assume  $\text{atm-of } L \in \text{atm-of ' (lits-of } M)$ 
      have  $g\text{-}r$ :  $\text{get-all-levels-of-marked } M = \text{rev [Suc } 0..<\text{Suc (backtrack-lvl } S)]$ 
        using  $\text{lev tr-}S$  unfolding  $\text{cdcl}_W\text{-}M\text{-level-inv-def}$  by auto
      have  $\text{Max (insert } 0 \text{ (set (get-all-levels-of-marked } M))) = (\text{backtrack-lvl } S)$ 
        unfolding  $g\text{-}r$  by (simp add:  $\text{Max-n-upt}$ )
      then have  $\text{get-level } L M = 0$ 
        using  $\text{get-maximum-possible-level-ge-get-level[of } L M]$ 
        unfolding  $\text{get-maximum-possible-level-max-get-all-levels-of-marked}$  by auto
    }
    ultimately show  $\text{get-level } L M = 0$  by blast
  qed
  then have ?case using  $\text{get-maximum-level-exists-lit-of-max-level[of } D \# \cup C M] \text{tr-}S T$ 
    by (auto simp:  $\text{Bex-mset-def}$ )
}
ultimately show ?case using  $\text{resolve.hyps}(3)$  by blast
next
case (skip  $L C' M D T$ ) note  $\text{tr-}S = \text{this}(1)$  and  $D = \text{this}(2)$  and  $T = \text{this}(5)$ 
then obtain  $La$  where  $La \in \# D$  and  $\text{get-level } La \text{ (Propagated } L C' \# M) = \text{backtrack-lvl } S$ 
  using  $\text{skip confl-inv}$  by auto
moreover
  have  $\text{atm-of } La \neq \text{atm-of } L$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have  $La$ :  $La = L$  using  $\langle La \in \# D \rangle \langle - L \notin \# D \rangle$  by (auto simp add:  $\text{atm-of-eq-atm-of}$ )
    have  $\text{Propagated } L C' \# M \models_{as} C \text{Not } D$ 
      using  $\text{conflicting tr-}S D$  unfolding  $\text{cdcl}_W\text{-conflicting-def}$  by auto
    then have  $-L \in \text{lits-of } M$ 
      using  $\langle La \in \# D \rangle \text{in-}C \text{Not-implies-uminus}(2)[\text{of } D L \text{Propagated } L C' \# M]$  unfolding  $La$ 
      by auto
  qed

```

```

    then show False using lev tr-S unfolding cdclW-M-level-inv-def consistent-interp-def by auto
  qed
  then have get-level La (Propagated L C' # M) = get-level La M by auto
  ultimately show ?case using D tr-S T by auto
qed (auto split: split-if-asm simp: cdclW-M-level-inv-decomp)

```

### 17.6.5 Strong completeness

**lemma** *cdcl<sub>W</sub>-cp-propagate-confl*:

```

  assumes cdclW-cp S T
  shows propagate** S T ∨ (∃ S'. propagate** S S' ∧ conflict S' T)
  using assms by induction blast+

```

**lemma** *rtrancpl-cdcl<sub>W</sub>-cp-propagate-confl*:

```

  assumes cdclW-cp** S T
  shows propagate** S T ∨ (∃ S'. propagate** S S' ∧ conflict S' T)
  by (simp add: assms rtrancpl-cdclW-cp-propa-or-propa-confl)

```

**lemma** *cdcl<sub>W</sub>-cp-propagate-completeness*:

```

  assumes MN: set M ⊨s set-mset N and
  cons: consistent-interp (set M) and
  tot: total-over-m (set M) (set-mset N) and
  lits-of (trail S) ⊆ set M and
  init-clss S = N and
  propagate** S S' and
  learned-clss S = {#}
  shows length (trail S) ≤ length (trail S') ∧ lits-of (trail S') ⊆ set M
  using assms(6,4,5,7)

```

**proof** (*induction rule: rtrancpl-induct*)

```

  case base
  then show ?case by auto

```

**next**

```

  case (step Y Z)
  note st = this(1) and propa = this(2) and IH = this(3) and lits' = this(4) and NS = this(5) and
  learned = this(6)
  then have len: length (trail S) ≤ length (trail Y) and LM: lits-of (trail Y) ⊆ set M
  by blast+

```

**obtain** *M' N' U k C L* **where**

```

  Y: state Y = (M', N', U, k, C-True) and
  Z: state Z = (Propagated L (C + {#L#}) # M', N', U, k, C-True) and
  C: C + {#L#} ∈# clauses Y and
  M'-C: M' ⊨as CNot C and
  undefined-lit (trail Y) L
  using propa by auto

```

**have** *init-clss S = init-clss Y*

using *st* by *induction auto*

**then have** [*simp*]: *N' = N* using *NS Y Z* by *simp*

**have** *learned-clss Y = {#}*

using *st learned* by *induction auto*

**then have** [*simp*]: *U = {#}* using *Y* by *auto*

**have** *set M ⊨<sub>s</sub> CNot C*

using *M'-C LM Y* unfolding *true-annots-def Ball-def true-annot-def true-clss-def true-cl-def*  
by *force*

**moreover**

**have** *set M ⊨ C + {#L#}*

**using**  $MN$   $C$  *learned*  $Y$  **unfolding** *true-clss-def clauses-def*  
**by** (*metis*  $NS$   $\langle \text{init-clss } S = \text{init-clss } Y \rangle \langle \text{learned-clss } Y = \{\#\} \rangle$  *add.right-neutral*  
*mem-set-mset-iff*)  
**ultimately have**  $L \in \text{set } M$  **by** (*simp add: cons consistent-CNot-not*)  
**then show**  $?case$  **using**  $LM$  *len*  $Y$   $Z$  **by** *auto*  
**qed**

**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*: *lits-of* (*trail*  $S$ )  $\subseteq \text{set } M$   
**shows**  $\exists S'. \text{propagate}^{**} S S' \wedge \text{full } \text{cdcl}_W\text{-cp } S S'$   
**proof** –  
**obtain**  $S'$  **where** *full*: *full*  $\text{cdcl}_W\text{-cp } S S'$   
**using** *always-exists-full-cdcl<sub>W</sub>-cp-step alien* **by** *blast*  
**then consider** (*propa*)  $\text{propagate}^{**} S S'$   
 $|$  (*confl*)  $\exists X. \text{propagate}^{**} S X \wedge \text{conflict } X S'$   
**using** *rtrancp-cdcl<sub>W</sub>-cp-propagate-confl* **unfolding** *full-def* **by** *blast*  
**then show**  $?thesis$   
**proof** *cases*  
**case** *propa* **then show**  $?thesis$  **using** *full* **by** *blast*  
**next**  
**case** *confl*  
**then obtain**  $X$  **where**  
 $X$ :  $\text{propagate}^{**} S X$  **and**  
 $X\text{conf}$ :  $\text{conflict } X S'$   
**by** *blast*  
**have** *clsX*: *init-clss*  $X = \text{init-clss } S$   
**using**  $X$  **by** *induction auto*  
**have** *learnedX*: *learned-clss*  $X = \{\#\}$  **using**  $X$  *learned* **by** *induction auto*  
**obtain**  $E$  **where**  
 $E$ :  $E \in \#$  *init-clss*  $X + \text{learned-clss } X$  **and**  
 $\text{Not-}E$ :  $\text{trail } X \models_{as} CNot E$   
**using**  $X\text{conf}$  **by** (*auto simp add: conflict.simps clauses-def*)  
**have** *lits-of* (*trail*  $X$ )  $\subseteq \text{set } M$   
**using** *cdcl<sub>W</sub>-cp-propagate-completeness*[*OF assms(1–3) lits - X learned*] *learned* **by** *auto*  
**then have**  $MNE$ :  $\text{set } M \models_s CNot E$   
**using**  $\text{Not-}E$   
**by** (*fastforce simp add: true-annots-def true-annot-def true-clss-def true-clss-def*)  
**have**  $\neg \text{set } M \models_s \text{set-mset } N$   
**using**  $E$  *consistent-CNot-not*[*OF cons MNE*]  
**unfolding** *learnedX true-clss-def* **unfolding** *clsX clsS* **by** *auto*  
**then show**  $?thesis$  **using**  $MN$  **by** *blast*  
**qed**  
**qed**

See also  $\text{cdcl}_W\text{-cp}^{**} ?S ?S' \implies \exists M. \text{trail } ?S' = M @ \text{trail } ?S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$

**lemma** *rtrancp-propagate-is-trail-append:*

$\text{propagate}^{**} S T \implies \exists c. \text{trail } T = c @ \text{trail } S$   
**by** (*induction rule: rtrancp-induct*) *auto*

**lemma** *rtrancp-propagate-is-update-trail*:  
 $\text{propagate}^{**} S T \implies \text{cdcl}_W\text{-}M\text{-level-inv } S \implies T \sim \text{delete-trail-and-rebuild } (\text{trail } T) S$   
**proof** (*induction rule: rtrancp-induct*)  
 case *base*  
 then show ?case **unfolding** *state-eq-def* **by** (*auto simp: cdcl<sub>W</sub>-M-level-inv-decomp*)  
**next**  
 case (*step T U*) **note**  $IH = \text{this}(3)[\text{OF } \text{this}(4)]$   
**moreover have** *cdcl<sub>W</sub>-M-level-inv U*  
 using *rtrancp-cdcl<sub>W</sub>-consistent-inv*  $\langle \text{propagate}^{**} S T \rangle \langle \text{propagate } T U \rangle$   
*rtrancp-mono*[*of propagate cdcl<sub>W</sub>*] *cdcl<sub>W</sub>-cp-consistent-inv propagate'*  
*rtrancp-propagate-is-rtrancp-cdcl<sub>W</sub> step.prem*s **by** *blast*  
 then have *no-dup (trail U)* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**ultimately show** ?case **using**  $\langle \text{propagate } T U \rangle$  **unfolding** *state-eq-def* **by** *fastforce*  
**qed**

**lemma** *cdcl<sub>W</sub>-stgy-strong-completeness-n*:

**assumes**

*MN*: *set M*  $\models_s$  *set-mset N* **and**  
*cons*: *consistent-interp (set M)* **and**  
*tot*: *total-over-m (set M) (set-mset N)* **and**  
*atm-incl*: *atm-of ' (set M)  $\subseteq$  atm-of-mu N* **and**  
*distM*: *distinct M* **and**  
*length*:  $n \leq \text{length } M$

**shows**

$\exists M' k S. \text{length } M' \geq n \wedge$   
*lits-of*  $M' \subseteq \text{set } M \wedge$   
*no-dup*  $M' \wedge$   
 $S \sim \text{update-backtrack-lvl } k (\text{append-trail } (\text{rev } M') (\text{init-state } N)) \wedge$   
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$

**using** *length*

**proof** (*induction n*)

case *0*

**have**  $\text{update-backtrack-lvl } 0 (\text{append-trail } (\text{rev } []) (\text{init-state } N)) \sim \text{init-state } N$   
**by** (*auto simp: state-eq-def simp del: state-simp*)

**moreover have**

$0 \leq \text{length } []$  **and**  
*lits-of*  $[] \subseteq \text{set } M$  **and**  
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) (\text{init-state } N)$   
**and** *no-dup []*  
**by** (*auto simp: state-eq-def simp del: state-simp*)

**ultimately show** ?case **using** *state-eq-sym* **by** *blast*

**next**

case (*Suc n*) **note**  $IH = \text{this}(1)$  **and**  $n = \text{this}(2)$

**then obtain**  $M' k S$  **where**

$l\text{-}M'$ :  $\text{length } M' \geq n$  **and**  
 $M'$ : *lits-of*  $M' \subseteq \text{set } M$  **and**  
 $n\text{-d}[simp]$ : *no-dup*  $M'$  **and**  
 $S$ :  $S \sim \text{update-backtrack-lvl } k (\text{append-trail } (\text{rev } M') (\text{init-state } N))$  **and**  
 $st$ :  $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$   
**by** *auto*

**have**

$M$ : *cdcl<sub>W</sub>-M-level-inv S* **and**  
*alien*: *no-strange-atm S*  
**using** *rtrancp-cdcl<sub>W</sub>-consistent-inv*[*OF rtrancp-cdcl<sub>W</sub>-stgy-rtrancp-cdcl<sub>W</sub>*][*OF st*]



```

    rtrancpl-cdclW-no-strange-atm-inv[OF rtrancpl-cdclW-stgy-rtrancpl-cdclW[OF st]]
    S unfolding state-eq-def cdclW-M-level-inv-def no-strange-atm-def by auto
  { assume no-step:  $\neg$ no-step propagate S

  obtain S' where S': propagate** S S' and full: full cdclW-cp S S'
    using completeness-is-a-full1-propagation[OF assms(1-3), of S] alien M' S by auto
  have lev: cdclW-M-level-inv S'
    using M S' rtrancpl-cdclW-consistent-inv rtrancpl-propagate-is-rtrancpl-cdclW by blast
  then have n-d'[simp]: no-dup (trail S')
    unfolding cdclW-M-level-inv-def by auto
  have length (trail S)  $\leq$  length (trail S')  $\wedge$  lits-of (trail S')  $\subseteq$  set M
    using S' full cdclW-cp-propagate-completeness[OF assms(1-3), of S] M' S by auto
  moreover
    have full: full1 cdclW-cp S S'
      using full no-step no-step-cdclW-cp-no-conflict-no-propagate(2) unfolding full1-def full-def
      rtrancpl-unfold by blast
    then have cdclW-stgy S S' by (simp add: cdclW-stgy.conflict')
  moreover
    have propa: propagate++ S S' using S' full unfolding full1-def by (metis rtrancplD trancplD)
    have trail S = M' using S by auto
    with propa have length (trail S')  $>$  n
      using l-M' propa by (induction rule: trancpl.induct) auto
  moreover
    have stS': cdclW-stgy** (init-state N) S'
      using st cdclW-stgy.conflict'[OF full] by auto
    then have init-clss S' = N using stS' rtrancpl-cdclW-stgy-no-more-init-clss by fastforce
  moreover
    have
      [simp]: learned-clss S' = {#} and
      [simp]: init-clss S' = init-clss S and
      [simp]: conflicting S' = C-True
      using trancpl-into-rtrancpl[OF  $\langle$ propagate++ S S' $\rangle$ ] S
      rtrancpl-propagate-is-update-trail[of S S'] S M unfolding state-eq-def by simp-all
    have S-S': S'  $\sim$  update-backtrack-lvl (backtrack-lvl S')
      (append-trail (rev (trail S')) (init-state N)) using S
      by (auto simp: state-eq-def simp del: state-simp)
    have cdclW-stgy** (init-state (init-clss S')) S'
      apply (rule rtrancpl.rtrancpl-into-rtrancpl)
      using st unfolding  $\langle$ init-clss S' = N $\rangle$  apply simp
      using  $\langle$ cdclW-stgy S S' $\rangle$  by simp
  ultimately have ?case
    apply -
    apply (rule exI[of - trail S'], rule exI[of - backtrack-lvl S'], rule exI[of - S'])
    using S-S' by (auto simp: state-eq-def simp del: state-simp)
  }
  moreover {
    assume no-step: no-step propagate S
    have ?case
      proof (cases length M'  $\geq$  Suc n)
        case True
          then show ?thesis using l-M' M' st M alien S by fastforce
        next
          case False
            then have n': length M' = n using l-M' by auto
            have no-conf: no-step conflict S

```

```

proof –
  { fix  $D$ 
    assume  $D \in \# N$  and  $M' \models_{as} CNot\ D$ 
    then have  $set\ M \models D$  using  $MN$  unfolding  $true-clss-def$  by  $auto$ 
    moreover have  $set\ M \models_s CNot\ D$ 
      using  $\langle M' \models_{as} CNot\ D \rangle M'$ 
      by  $(metis\ le-iff-sup\ true-annots-true-cl\ true-clss-union-increase)$ 
    ultimately have  $False$  using  $cons\ consistent-CNot-not$  by  $blast$ 
  }
  then show  $?thesis$  using  $S$  by  $(auto\ simp\ add:\ conflict.simps\ true-clss-def)$ 
qed
have  $lenM: length\ M = card\ (set\ M)$  using  $distM$  by  $(induction\ M)\ auto$ 
have  $no-dup\ M'$  using  $S\ M$  unfolding  $cdcl_W-M-level-inv-def$  by  $auto$ 
then have  $card\ (lits-of\ M') = length\ M'$ 
  by  $(induction\ M')\ (auto\ simp\ add:\ lits-of-def\ card-insert-if)$ 
then have  $lits-of\ M' \subseteq set\ M$ 
  using  $n\ M'\ n'\ lenM$  by  $auto$ 
then obtain  $m$  where  $m: m \in set\ M$  and  $undef-m: m \notin lits-of\ M'$  by  $auto$ 
moreover have  $undef: undefined-lit\ M'\ m$ 
  using  $M'\ Marked-Propagated-in-iff-in-lits-of\ calculation(1,2)\ cons$ 
   $consistent-interp-def$  by  $blast$ 
moreover have  $atm-of\ m \in atms-of-mu\ (init-clss\ S)$ 
  using  $atm-incl\ calculation\ S$  by  $auto$ 
ultimately
  have  $dec: decide\ S\ (cons-trail\ (Marked\ m\ (k+1))\ (incr-lvl\ S))$ 
    using  $decide.intros[of\ S\ rev\ M'\ N - k\ m$ 
       $cons-trail\ (Marked\ m\ (k + 1))\ (incr-lvl\ S)]\ S$ 
    by  $auto$ 
let  $?S' = cons-trail\ (Marked\ m\ (k+1))\ (incr-lvl\ S)$ 
have  $lits-of\ (trail\ ?S') \subseteq set\ M$  using  $m\ M'\ S\ undef$  by  $auto$ 
moreover have  $no-strange-atm\ ?S'$ 
  using  $alien\ dec\ M$  by  $(meson\ cdcl_W-no-strange-atm-inv\ decide\ other)$ 
ultimately obtain  $S''$  where  $S'': propagate^{**}\ ?S'\ S''$  and  $full: full\ cdcl_W-cp\ ?S'\ S''$ 
  using  $completeness-is-a-full1-propagation[OF\ assms(1-3),\ of\ ?S']\ S\ undef$  by  $auto$ 
have  $cdcl_W-M-level-inv\ ?S'$ 
  using  $M\ dec\ rtranclp-mono[of\ decide\ cdcl_W]$  by  $(meson\ cdcl_W-consistent-inv\ decide\ other)$ 
then have  $lev'': cdcl_W-M-level-inv\ S''$ 
  using  $S''\ rtranclp-cdcl_W-consistent-inv\ rtranclp-propagate-is-rtranclp-cdcl_W$  by  $blast$ 
then have  $n-d'': no-dup\ (trail\ S'')$ 
  unfolding  $cdcl_W-M-level-inv-def$  by  $auto$ 
have  $length\ (trail\ ?S') \leq length\ (trail\ S'') \wedge lits-of\ (trail\ S'') \subseteq set\ M$ 
  using  $S''\ full\ cdcl_W-cp-propagate-completeness[OF\ assms(1-3),\ of\ ?S'\ S'']\ m\ M'\ S\ undef$ 
  by  $simp$ 
then have  $Suc\ n \leq length\ (trail\ S'') \wedge lits-of\ (trail\ S'') \subseteq set\ M$ 
  using  $l-M'\ S\ undef$  by  $auto$ 
moreover
  have  $cdcl_W-M-level-inv\ (cons-trail\ (Marked\ m\ (Suc\ (backtrack-lvl\ S))))$ 
     $(update-backtrack-lvl\ (Suc\ (backtrack-lvl\ S))\ S))$ 
    using  $S\ \langle cdcl_W-M-level-inv\ (cons-trail\ (Marked\ m\ (k + 1))\ (incr-lvl\ S)) \rangle$  by  $auto$ 
then have  $S'': S'' \sim update-backtrack-lvl\ (backtrack-lvl\ S'')$ 
     $(append-trail\ (rev\ (trail\ S''))\ (init-state\ N))$ 
    using  $rtranclp-propagate-is-update-trail[OF\ S'']\ S\ undef\ n-d''\ lev''$ 
    by  $(auto\ simp\ del:\ state-simp\ simp:\ state-eq-def)$ 
then have  $cdcl_W-stgy^{**}\ (init-state\ N)\ S''$ 
  using  $cdcl_W-stgy.intros(2)[OF\ decide[OF\ dec] - full]\ no-step\ no-conf\ st$ 

```

```

      by (auto simp: cdclW-cp.simps)
      ultimately show ?thesis using S'' n-d'' by blast
    qed
  }
  ultimately show ?case by blast
qed

lemma cdclW-stgy-strong-completeness:
  assumes MN: set M ⊨s set-mset N
  and cons: consistent-interp (set M)
  and tot: total-over-m (set M) (set-mset N)
  and atm-incl: atm-of ' (set M) ⊆ atms-of-mu N
  and distM: distinct M
  shows
    ∃ M' k S.
      lits-of M' = set M ∧
      S ∼ update-backtrack-lvl k (append-trail (rev M') (init-state N)) ∧
      cdclW-stgy** (init-state N) S ∧
      final-cdclW-state S
proof -
  from cdclW-stgy-strong-completeness-n[OF assms, of length M]
  obtain M' k T where
    l: length M ≤ length M' and
    M'-M: lits-of M' ⊆ set M and
    no-dup: no-dup M' and
    T: T ∼ update-backtrack-lvl k (append-trail (rev M') (init-state N)) and
    st: cdclW-stgy** (init-state N) T
  by auto
  have card (set M) = length M using distM by (simp add: distinct-card)
  moreover
    have cdclW-M-level-inv T
      using rtranclp-cdclW-stgy-consistent-inv[OF st] T by auto
    then have card (set ((map (λl. atm-of (lit-of l)) M'))) = length M'
      using distinct-card no-dup by fastforce
  moreover have card (lits-of M') = card (set ((map (λl. atm-of (lit-of l)) M')))
    using no-dup unfolding lits-of-def apply (induction M') by (auto simp add: card-insert-if)
  ultimately have card (set M) ≤ card (lits-of M') using l unfolding lits-of-def by auto
  then have set M = lits-of M'
    using M'-M card-seteq by blast
  moreover
    then have M' ⊨asm N
      using MN unfolding true-annots-def Ball-def true-annot-def true-clss-def by auto
    then have final-cdclW-state T
      using T no-dup unfolding final-cdclW-state-def by auto
  ultimately show ?thesis using st T by blast
qed

```

### 17.6.6 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* ( $S::'st$ )  $\equiv$   
 $(\forall M K i M' D. M' @ \text{Marked } K \ i \ \# \ M = \text{trail } S \longrightarrow D \in \# \text{ clauses } S$   
 $\longrightarrow \neg M \models_{as} CNot \ D)$

```

lemma no-smaller-confl-init-sate[simp]:
  no-smaller-confl (init-state N) unfolding no-smaller-confl-def by auto

lemma cdclW-o-no-smaller-confl-inv:
  fixes S S' :: 'st
  assumes
    cdclW-o S S' and
    lev: cdclW-M-level-inv S and
    max-lev: conflict-is-false-with-level S and
    smaller: no-smaller-confl S and
    no-f: no-clause-is-false S
  shows no-smaller-confl S'
  using assms(1,2) unfolding no-smaller-confl-def
proof (induct rule: cdclW-o-induct-lev2)
case (decide L T) note confl = this(1) and undef = this(2) and T = this(4)
have [simp]: clauses T = clauses S
  using T undef by auto
show ?case
proof (intro allI impI)
  fix M'' K i M' Da
  assume M'' @ Marked K i # M' = trail T
  and D: Da ∈ # local.clauses T
  then have tl M'' @ Marked K i # M' = trail S
    ∨ (M'' = [] ∧ Marked K i # M' = Marked L (backtrack-lvl S + 1) # trail S)
    using T undef by (cases M'') auto
  moreover {
    assume tl M'' @ Marked K i # M' = trail S
    then have ¬M' ⊨as CNot Da
      using D T undef no-f confl smaller unfolding no-smaller-confl-def smaller by fastforce
  }
  moreover {
    assume Marked K i # M' = Marked L (backtrack-lvl S + 1) # trail S
    then have ¬M' ⊨as CNot Da using no-f D confl T by auto
  }
  ultimately show ¬M' ⊨as CNot Da by fast
qed
next
case resolve
then show ?case using smaller no-f max-lev unfolding no-smaller-confl-def by auto
next
case skip
then show ?case using smaller no-f max-lev unfolding no-smaller-confl-def by auto
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and confl = this(3) and undef = this(6)
  and T = this(7)
obtain c where M: trail S = c @ M2 @ Marked K (i+1) # M1
  using decomp by auto

show ?case
proof (intro allI impI)
  fix M ia K' M' Da
  assume M' @ Marked K' ia # M = trail T
  then have tl M' @ Marked K' ia # M = M1
    using T decomp undef lev by (cases M') (auto simp: cdclW-M-level-inv-decomp)
  assume D: Da ∈ # clauses T

```

```

moreover{
  assume  $Da \in \# \text{ clauses } S$ 
  then have  $\neg M \models_{as} CNot \ Da$  using  $\langle tl \ M' @ \text{Marked } K' \text{ ia } \# \ M = M1 \rangle \ M \text{ confl undef smaller}$ 
  unfolding no-smaller-conflict-def by auto
}
moreover {
  assume  $Da: Da = D + \{\#L\# \}$ 
  have  $\neg M \models_{as} CNot \ Da$ 
  proof (rule ccontr)
    assume  $\neg \text{?thesis}$ 
    then have  $\neg L \in \text{lits-of } M$  unfolding  $Da$  by auto
    then have  $\neg L \in \text{lits-of } (Propagated \ L \ ((D + \{\#L\# \})) \ \# \ M1)$ 
      using  $UnI2 \ \langle tl \ M' @ \text{Marked } K' \text{ ia } \# \ M = M1 \rangle$ 
      by auto
    moreover
      have backtrack S
        (cons-trail (Propagated L ( $D + \{\#L\# \}$ ))
          (reduce-trail-to M1 (add-learned-cls ( $D + \{\#L\# \}$ )
            (update-backtrack-lvl i (update-conflicting C-True S))))))
        using backtrack.intros[of S] backtrack.hyps
        by (force simp: state-eq-def simp del: state-simp)
      then have cdclW-M-level-inv
        (cons-trail (Propagated L ( $D + \{\#L\# \}$ ))
          (reduce-trail-to M1 (add-learned-cls ( $D + \{\#L\# \}$ )
            (update-backtrack-lvl i (update-conflicting C-True S))))))
        using cdclW-consistent-inv[OF - lev] other[OF bj] by auto
      then have no-dup (Propagated L ( $D + \{\#L\# \}$ )  $\# \ M1$ )
        using decomp undef lev unfolding cdclW-M-level-inv-def by auto
      ultimately show False by (metis consistent-interp-def distinctconsistent-interp
        insertCI lits-of-cons marked-lit.sel(2))
    qed
  }
  ultimately show  $\neg M \models_{as} CNot \ Da$ 
    using  $T \text{ undef } \langle Da = D + \{\#L\# \} \implies \neg M \models_{as} CNot \ Da \rangle \text{ decomp lev}$ 
    unfolding cdclW-M-level-inv-def by fastforce
  qed
qed

```

**lemma** *conflict-no-smaller-conflict-inv*:

```

assumes conflict S S'
and no-smaller-conflict S
shows no-smaller-conflict S'
using assms unfolding no-smaller-conflict-def by fastforce

```

**lemma** *propagate-no-smaller-conflict-inv*:

```

assumes propagate: propagate S S'
and n-l: no-smaller-conflict S
shows no-smaller-conflict S'
unfolding no-smaller-conflict-def
proof (intro allI impI)
  fix  $M' \ K \ i \ M'' \ D$ 
  assume  $M': M'' @ \text{Marked } K \ i \ \# \ M' = \text{trail } S'$ 
  and  $D \in \# \text{ clauses } S'$ 
  obtain  $M \ N \ U \ k \ C \ L$  where
     $S: \text{state } S = (M, N, U, k, C\text{-True})$  and

```

$S'$ : state  $S' = (\text{Propagated } L \ ( (C + \{\#L\#\}) \# M, N, U, k, C\text{-True}) \text{ and } C + \{\#L\#\} \in \# \text{ clauses } S \text{ and } M \models_{as} C \text{Not } C \text{ and } \text{undefined-lit } M \ L$   
**using** *propagate* **by** *auto*  
**have**  $tl \ M'' @ \text{Marked } K \ i \ \# \ M' = \text{trail } S \text{ using } M' \ S'$   
**by** (*metis Pair-inject list.inject list.sel(3) marked-lit.distinct(1) self-append-conv2 tl-append2*)  
**then have**  $\neg M' \models_{as} C \text{Not } D$   
**using**  $\langle D \in \# \text{ clauses } S' \rangle \ n\text{-l } S \ S' \text{ clauses-def unfolding no-smaller-confli-def}$  **by** *auto*  
**then show**  $\neg M' \models_{as} C \text{Not } D$  **by** *auto*  
**qed**

**lemma** *cdcl<sub>W</sub>-cp-no-smaller-confli-inv*:  
**assumes** *propagate: cdcl<sub>W</sub>-cp S S'*  
**and** *n-l: no-smaller-confli S*  
**shows** *no-smaller-confli S'*  
**using** *assms*  
**proof** (*induct rule: cdcl<sub>W</sub>-cp.induct*)  
**case** (*conflict' S S'*)  
**then show** *?case* **using** *conflict-no-smaller-confli-inv[of S S']* **by** *blast*  
**next**  
**case** (*propagate' S S'*)  
**then show** *?case* **using** *propagate-no-smaller-confli-inv[of S S']* **by** *fastforce*  
**qed**

**lemma** *rtranclp-cdcl<sub>W</sub>-cp-no-smaller-confli-inv*:  
**assumes** *propagate: cdcl<sub>W</sub>-cp\*\* S S'*  
**and** *n-l: no-smaller-confli S*  
**shows** *no-smaller-confli S'*  
**using** *assms*  
**proof** (*induct rule: rtranclp-induct*)  
**case** *base*  
**then show** *?case* **by** *simp*  
**next**  
**case** (*step S' S''*)  
**then show** *?case* **using** *cdcl<sub>W</sub>-cp-no-smaller-confli-inv[of S' S'']* **by** *fast*  
**qed**

**lemma** *tranclp-cdcl<sub>W</sub>-cp-no-smaller-confli-inv*:  
**assumes** *propagate: cdcl<sub>W</sub>-cp<sup>++</sup> S S'*  
**and** *n-l: no-smaller-confli S*  
**shows** *no-smaller-confli S'*  
**using** *assms*  
**proof** (*induct rule: tranclp.induct*)  
**case** (*r-into-trancl S S'*)  
**then show** *?case* **using** *cdcl<sub>W</sub>-cp-no-smaller-confli-inv[of S S']* **by** *blast*  
**next**  
**case** (*trancl-into-trancl S S' S''*)  
**then show** *?case* **using** *cdcl<sub>W</sub>-cp-no-smaller-confli-inv[of S' S'']* **by** *fast*  
**qed**

**lemma** *full-cdcl<sub>W</sub>-cp-no-smaller-confli-inv*:  
**assumes** *full cdcl<sub>W</sub>-cp S S'*  
**and** *n-l: no-smaller-confli S*

**shows** *no-smaller-conf*  $S'$   
**using** *assms* **unfolding** *full-def*  
**using** *rtrancp-cdcl<sub>W</sub>-cp-no-smaller-conf-inv*[*of S S'*] **by** *blast*

**lemma** *full1-cdcl<sub>W</sub>-cp-no-smaller-conf-inv*:  
**assumes** *full1 cdcl<sub>W</sub>-cp S S'*  
**and** *n-l: no-smaller-conf S*  
**shows** *no-smaller-conf S'*  
**using** *assms* **unfolding** *full1-def*  
**using** *trancp-cdcl<sub>W</sub>-cp-no-smaller-conf-inv*[*of S S'*] **by** *blast*

**lemma** *cdcl<sub>W</sub>-stgy-no-smaller-conf-inv*:  
**assumes** *cdcl<sub>W</sub>-stgy S S'*  
**and** *n-l: no-smaller-conf S*  
**and** *conflict-is-false-with-level S*  
**and** *cdcl<sub>W</sub>-M-level-inv S*  
**shows** *no-smaller-conf S'*  
**using** *assms*  
**proof** (*induct rule: cdcl<sub>W</sub>-stgy.induct*)  
**case** (*conflict' S'*)  
**then show** ?*case* **using** *full1-cdcl<sub>W</sub>-cp-no-smaller-conf-inv*[*of S S'*] **by** *blast*  
**next**  
**case** (*other' S' S''*)  
**have** *no-smaller-conf S'*  
**using** *cdcl<sub>W</sub>-o-no-smaller-conf-inv*[*OF other'.hyps(1) other'.prems(3,2,1)*]  
*not-conflict-not-any-negated-init-clss other'.hyps(2)* **by** *blast*  
**then show** ?*case* **using** *full-cdcl<sub>W</sub>-cp-no-smaller-conf-inv*[*of S' S''*] *other'.hyps* **by** *blast*  
**qed**

**lemma** *conflict-conflict-is-no-clause-is-false-test*:  
**assumes** *conflict S S'*  
**and**  $(\forall D \in \# \text{init-clss } S + \text{learned-clss } S. \text{trail } S \models_{\text{as}} \text{CNot } D$   
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S))$   
**shows**  $\forall D \in \# \text{init-clss } S' + \text{learned-clss } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$   
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S')$   
**using** *assms* **by** *auto*

**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' = C-True*  
**shows**  $\exists S''. \text{conflict } S' S''$   
**using** *assms* *clauses-def not-conflict-not-any-negated-init-clss* **by** *fastforce*

**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-conf S*  
**shows** *no-clause-is-false S'*  
 $\vee (\text{conflicting } S' = \text{C-True}$   
 $\longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$

```

      → (∃ L. L ∈# D ∧ get-level L (trail S') = backtrack-lvl S'))
using assms(1,2)
proof (induct rule: cdelw-o-induct-lev2)
case (decide L T) note S = this(1) and undef = this(2) and T = this(4)
show ?case
proof (rule HOL.disjI2, clarify)
fix D
assume D: D ∈# clauses T and M-D: trail T ⊨as CNot D
let ?M = trail S
let ?M' = trail T
let ?k = backtrack-lvl S
have ¬?M ⊨as CNot D
  using no-f D S T undef by auto
have ¬L ∈# D
proof (rule ccontr)
assume ¬ ?thesis
have ?M ⊨as CNot D
  unfolding true-annots-def Ball-def true-annot-def CNot-def true-cls-def
proof (intro allI impI)
fix x
assume x: x ∈ { {#- L#} | L. L ∈# D }

  then obtain L' where L': x = {#- L'#} L' ∈# D by auto
  obtain L'' where L'' ∈# x and lits-of (Marked L (?k + 1) # ?M) ⊨l L''
    using M-D x T undef unfolding true-annots-def Ball-def true-annot-def CNot-def
    true-cls-def Bex-mset-def by auto
  show ∃ L ∈# x. lits-of ?M ⊨l L unfolding Bex-mset-def
    by (metis ⟨- L ∉# D⟩ ⟨L'' ∈# x⟩ L' ⟨lits-of (Marked L (?k + 1) # ?M) ⊨l L'⟩
        count-single insertE less-numeral-extra(3) lits-of-cons marked-lit.sel(1)
        true-lit-def uminus-of-uminus-id)
qed
  then show False using ⟨¬ ?M ⊨as CNot D⟩ by auto
qed
have atm-of L ∉ atm-of ' (lits-of ?M)
  using undef defined-lit-map unfolding lits-of-def by fastforce
then have get-level (¬L) (Marked L (?k + 1) # ?M) = ?k + 1 by simp
then show ∃ La. La ∈# D ∧ get-level La ?M'
  = backtrack-lvl T
  using ⟨¬L ∈# D⟩ T undef by auto
qed
next
case resolve
then show ?case by auto
next
case skip
then show ?case by auto
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and undef = this(6) and T = this(7)
show ?case
proof (rule HOL.disjI2, clarify)
fix Da
assume Da: Da ∈# clauses T
and M-D: trail T ⊨as CNot Da
obtain c where M: trail S = c @ M2 @ Marked K (i + 1) # M1
  using decomp by auto

```



```

have tr-T: trail T = Propagated L (D + {#L#}) # M1
  using T decomp undef lev by (auto simp: cdclW-M-level-inv-decomp)
have backtrack S T
  using backtrack.intros backtrack.hyps T by (force simp del: state-simp simp: state-eq-def)
then have lev': cdclW-M-level-inv T
  using cdclW-consistent-inv lev other by blast
then have - L ∉ lits-of M1
  unfolding cdclW-M-level-inv-def lits-of-def
  proof -
    have consistent-interp (lits-of (trail S)) ∧ no-dup (trail S)
      ∧ backtrack-lvl S = length (get-all-levels-of-marked (trail S))
      ∧ get-all-levels-of-marked (trail S)
        = rev [1..1 + length (get-all-levels-of-marked (trail S))]
    using lev cdclW-M-level-inv-def by blast
  then show - L ∉ lit-of ' set M1
    by (metis (no-types) One-nat-def add.right-neutral add-Suc-right
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set backtrack.hyps(2)
      cdclW-ops.backtrack-lit-skipped cdclW-ops-axioms decomp lits-of-def)
  qed
{ assume Da ∈# clauses S
  then have ¬M1 ⊨as CNot Da using no-l M unfolding no-smaller-conflict-def by auto
}
moreover {
  assume Da: Da = D + {#L#}
  have ¬M1 ⊨as CNot Da using ⟨- L ∉ lits-of M1⟩ unfolding Da by simp
}
ultimately have ¬M1 ⊨as CNot Da
  using Da T undef decomp lev by (fastforce simp: cdclW-M-level-inv-decomp)
then have -L ∈# Da
  using M-D ⟨- L ∉ lits-of M1⟩ in-CNot-implies-uminus(2)
  true-annots-CNot-lit-of-notin-skip T unfolding tr-T
  by (smt insert-iff lits-of-cons marked-lit.sel(2))
have g-M1: get-all-levels-of-marked M1 = rev [1..i+1]
  using lev lev' T decomp undef unfolding cdclW-M-level-inv-def by auto
have no-dup (Propagated L (D + {#L#}) # M1)
  using lev lev' T decomp undef unfolding cdclW-M-level-inv-def by auto
then have L: atm-of L ∉ atm-of ' lits-of M1 unfolding lits-of-def by auto
have get-level (-L) (Propagated L ((D + {#L#})) # M1) = i
  using get-level-get-rev-level-get-all-levels-of-marked[OF L,
    of [Propagated L ((D + {#L#}))]]
  by (simp add: g-M1 split: if-splits)
then show ∃ La. La ∈# Da ∧ get-level La (trail T) = backtrack-lvl T
  using ⟨-L ∈# Da⟩ T decomp undef lev by (auto simp: cdclW-M-level-inv-def)
qed
qed

lemma full1-cdclW-cp-exists-conflict-decompose:
  assumes confl: ∃ D ∈# clauses S. trail S ⊨as CNot D
  and full: full cdclW-cp S U
  and no-conf: conflicting S = C-True
  shows ∃ T. propagate** S T ∧ conflict T U
proof -
  consider (propa) propagate** S U
  | (confl) T where propagate** S T and conflict T U
  using full unfolding full-def by (blast dest: rtranclp-cdclW-cp-propa-or-propa-conf)

```

```

then show ?thesis
proof cases
  case confl
  then show ?thesis by blast
next
case propa
then have conflicting  $U = C\text{-True}$ 
  using no-confl by induction auto
moreover have [simp]: learned-clss  $U = \text{learned-clss } S$  and
  [simp]: init-clss  $U = \text{init-clss } S$ 
  using propa by induction auto
moreover
  obtain  $D$  where  $D: D \in \# \text{clauses } U$  and
     $\text{tr}S: \text{trail } S \models_{\text{as}} C\text{Not } D$ 
    using confl clauses-def by auto
  obtain  $M$  where  $M: \text{trail } U = M @ \text{trail } S$ 
    using full rtrancp-cdclW-cp-dropWhile-trail unfolding full-def by meson
  have  $\text{tr-}U: \text{trail } U \models_{\text{as}} C\text{Not } D$ 
    apply (rule true-annots-mono)
    using trS unfolding  $M$  by simp-all
  have  $\exists V. \text{conflict } U V$ 
    using (conflicting  $U = C\text{-True}$ )  $D$  clauses-def not-conflict-not-any-negated-init-clss  $\text{tr-}U$ 
    by blast
  then have False using full cdclW-cp.conflict' unfolding full-def by blast
  then show ?thesis by fast
qed
qed

```

**lemma** *full1-cdcl<sub>W</sub>-cp-exists-conflict-full1-decompose*:

```

assumes confl:  $\exists D \in \# \text{clauses } S. \text{trail } S \models_{\text{as}} C\text{Not } D$ 
and full: full cdclW-cp  $S U$ 
and no-confl: conflicting  $S = C\text{-True}$ 
shows  $\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U$ 
 $\wedge \text{trail } T \models_{\text{as}} C\text{Not } D \wedge \text{conflicting } U = C\text{-Clause } D \wedge D \in \# \text{clauses } S$ 

```

**proof** –

```

obtain  $T$  where propa:  $\text{propagate}^{**} S T$  and confl:  $\text{conflict } T U$ 
  using full1-cdclW-cp-exists-conflict-decompose[OF assms] by blast
have  $p: \text{learned-clss } T = \text{learned-clss } S \text{ init-clss } T = \text{init-clss } S$ 
  using propa by induction auto
have  $c: \text{learned-clss } U = \text{learned-clss } T \text{ init-clss } U = \text{init-clss } T$ 
  using confl by induction auto
obtain  $D$  where  $\text{trail } T \models_{\text{as}} C\text{Not } D \wedge \text{conflicting } U = C\text{-Clause } D \wedge D \in \# \text{clauses } S$ 
  using conf  $p c$  by (fastforce simp: clauses-def)
then show ?thesis
  using propa confl by blast
qed

```

**lemma** *cdcl<sub>W</sub>-stgy-no-smaller-confl*:

```

assumes cdclW-stgy  $S S'$ 
and n-l: no-smaller-confl  $S$ 
and conflict-is-false-with-level  $S$ 
and cdclW-M-level-inv  $S$ 
and no-clause-is-false  $S$ 
and distinct-cdclW-state  $S$ 
and cdclW-conflicting  $S$ 

```

```

shows no-smaller-confl S'
using assms
proof (induct rule: cdclW-stgy.induct)
case (conflict' S')
show no-smaller-confl S'
  using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-confl-inv by blast
next
case (other' S' S'')
have lev': cdclW-M-level-inv S'
  using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
show no-smaller-confl S''
  using cdclW-stgy-no-smaller-confl-inv[OF cdclW-stgy.other'[OF other'.hyps(1-3)]]
  other'.prems(1-3) by blast
qed

lemma cdclW-stgy-ex-lit-of-max-level:
assumes cdclW-stgy S S'
and n-l: no-smaller-confl S
and conflict-is-false-with-level S
and cdclW-M-level-inv S
and no-clause-is-false S
and distinct-cdclW-state S
and cdclW-conflicting S
shows conflict-is-false-with-level S'
using assms
proof (induct rule: cdclW-stgy.induct)
case (conflict' S')
have no-smaller-confl S'
  using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-confl-inv by blast
moreover have conflict-is-false-with-level S'
  using conflict'.hyps conflict'.prems(2-4)
  rtranclp-cdclW-co-conflict-ex-lit-of-max-level[of S S']
  unfolding full-def full1-def rtranclp-unfold by blast
then show ?case by blast
next
case (other' S' S'')
have lev': cdclW-M-level-inv S'
  using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
moreover
have no-clause-is-false S'
  ∨ (conflicting S' = C-True  $\longrightarrow$  ( $\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{as} C \text{Not } D$ 
 $\longrightarrow$  ( $\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S'$ )))
  using cdclW-o-conflict-is-no-clause-is-false[of S S'] other'.hyps(1) other'.prems(1-4) by fast
moreover {
  assume no-clause-is-false S'
  {
    assume conflicting S' = C-True
    then have conflict-is-false-with-level S' by auto
    moreover have full cdclW-cp S' S''
      by (metis (no-types) other'.hyps(3))
    ultimately have conflict-is-false-with-level S''
      using rtranclp-cdclW-co-conflict-ex-lit-of-max-level[of S' S''] lev' (no-clause-is-false S')
      by blast
  }
}
moreover

```

```

{
  assume c: conflicting  $S' \neq C\text{-True}$ 
  have conflicting  $S \neq C\text{-True}$  using other'.hyps(1) c
    by (induct rule: cdclW-o-induct) auto
  then have conflict-is-false-with-level  $S'$ 
    using cdclW-o-conflict-is-false-with-level-inv[OF other'.hyps(1)]
    other'.prems(3,5,6,2) by blast
  moreover have cdclW-cp**  $S' S''$  using other'.hyps(3) unfolding full-def by auto
  then have  $S' = S''$  using c
    by (induct rule: rtranclp-induct)
    (fastforce intro: conflicting-clause.exhaust)+
  ultimately have conflict-is-false-with-level  $S''$  by auto
}
ultimately have conflict-is-false-with-level  $S''$  by blast
}
moreover {
  assume confl: conflicting  $S' = C\text{-True}$ 
  and D-L:  $\forall D \in \# \text{ clauses } S'. \text{ trail } S' \models_{as} C\text{Not } D$ 
     $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S')$ 
  { assume  $\forall D \in \# \text{ clauses } S'. \neg \text{trail } S' \models_{as} C\text{Not } D$ 
    then have no-clause-is-false  $S'$  using  $\langle \text{conflicting } S' = C\text{-True} \rangle$  by simp
    then have conflict-is-false-with-level  $S''$  using calculation(3) by blast
  }
  moreover {
    assume  $\neg(\forall D \in \# \text{ clauses } S'. \neg \text{trail } S' \models_{as} C\text{Not } D)$ 
    then obtain T D where
      propagate**  $S' T$  and
      conflict  $T S''$  and
      D:  $D \in \# \text{ clauses } S'$  and
      trail  $S'' \models_{as} C\text{Not } D$  and
      conflicting  $S'' = C\text{-Clause } D$ 
      using full1-cdclW-cp-exists-conflict-full1-decompose[OF - -  $\langle \text{conflicting } S' = C\text{-True} \rangle$ ]
      other'(3) by (metis (mono-tags, lifting) ball-msetI bex-msetI conflictE state-eq-trail
        trail-update-conflicting)
    obtain M where M: trail  $S'' = M @ \text{trail } S'$  and nm:  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
      using rtranclp-cdclW-cp-dropWhile-trail other'(3) unfolding full-def by meson
    have btS: backtrack-lvl  $S'' = \text{backtrack-lvl } S'$ 
      using other'.hyps(3) unfolding full-def by (metis rtranclp-cdclW-cp-backtrack-lvl)
    have inv: cdclW-M-level-inv  $S''$ 
      by (metis (no-types) cdclW-stgy.conflict' cdclW-stgy-consistent-inv full-unfold lev'
        other'.hyps(3))
    then have nd: no-dup (trail  $S''$ )
      by (metis (no-types) cdclW-M-level-inv-decomp(2))
    have conflict-is-false-with-level  $S''$ 
    proof cases
      assume trail  $S' \models_{as} C\text{Not } D$ 
      moreover then obtain L where  $L \in \# D$  and get-level  $L (\text{trail } S') = \text{backtrack-lvl } S'$ 
        using D-L D by blast
      moreover
        have LS':  $-L \in \text{lits-of } (\text{trail } S')$ 
          using  $\langle \text{trail } S' \models_{as} C\text{Not } D \rangle \langle L \in \# D \rangle \text{in-}C\text{Not-implies-uminus}(2)$  by blast
        { fix x ::  $('v, \text{nat}, 'v \text{ literal multiset}) \text{ marked-lit}$  and
          xb ::  $('v, \text{nat}, 'v \text{ literal multiset}) \text{ marked-lit}$ 
          assume a1:  $x \in \text{set } (\text{trail } S')$  and
            a2: xb  $\in \text{set } M$  and

```

```

    a3: (λl. atm-of (lit-of l)) ‘ set M ∩ (λl. atm-of (lit-of l)) ‘ set (trail S')
    = {} and
    a4: - L = lit-of x and
    a5: atm-of L = atm-of (lit-of xb)
  moreover have atm-of (lit-of x) = atm-of L
    using a4 by (metis (no-types) atm-of-uminus)
  ultimately have False
    using a5 a3 a2 a1 by auto
}
then have atm-of L ∉ atm-of ‘ lits-of M
  using nd LS' unfolding M by (auto simp add: lits-of-def)
then have get-level L (trail S'') = get-level L (trail S')
  unfolding M by (simp add: lits-of-def)
ultimately show ?thesis using btS ⟨conflicting S'' = C-Clause D⟩ by auto
next
assume ¬trail S' ⊨as CNot D
then obtain L where L ∈ # D and LM: -L ∈ lits-of M
  using ⟨trail S'' ⊨as CNot D⟩
  by (auto simp add: CNot-def true-cls-def M true-annot-def true-annot-def
    split: split-if-asm)
{ fix x :: ('v, nat, 'v literal multiset) marked-lit and
  xb :: ('v, nat, 'v literal multiset) marked-lit
  assume a1: xb ∈ set (trail S') and
    a2: x ∈ set M and
    a3: atm-of L = atm-of (lit-of xb) and
    a4: - L = lit-of x and
    a5: (λl. atm-of (lit-of l)) ‘ set M ∩ (λl. atm-of (lit-of l)) ‘ set (trail S')
    = {}
  moreover have atm-of (lit-of xb) = atm-of (- L)
    using a3 by simp
  ultimately have False
    by auto }
then have LS': atm-of L ∉ atm-of ‘ lits-of (trail S')
  using nd ⟨L ∈ # D⟩ LM unfolding M by (auto simp add: lits-of-def)
show ?thesis
proof cases
  assume ne: get-all-levels-of-marked (trail S') = []
  have backtrack-lvl S'' = 0
    using inv ne nm unfolding cdclW-M-level-inv-def M
    by (simp add: get-all-levels-of-marked-nil-iff-not-is-marked)
  moreover
  have a1: get-rev-level L 0 (rev M) = 0
    using nm by auto
  then have get-level L (M @ trail S') = 0
    by (metis LS' get-all-levels-of-marked-nil-iff-not-is-marked
      get-level-skip-beginning-not-marked lits-of-def ne)
  ultimately show ?thesis using ⟨conflicting S'' = C-Clause D⟩ ⟨L ∈ # D⟩ unfolding M
    by auto
next
  assume ne: get-all-levels-of-marked (trail S') ≠ []
  have hd (get-all-levels-of-marked (trail S')) = backtrack-lvl S'
    using ne lev' M nm unfolding cdclW-M-level-inv-def
    by (cases get-all-levels-of-marked (trail S'))
    (simp-all add: get-all-levels-of-marked-nil-iff-not-is-marked[symmetric])
  moreover have atm-of L ∈ atm-of ‘ lits-of M

```

```

      using  $\langle \neg L \in \text{ lits-of } M \rangle$ 
      by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def)
    ultimately show ?thesis
      using nm ne  $\langle L \in \#D \rangle$   $\langle \text{conflicting } S'' = C\text{-Clause } D \rangle$ 
        get-level-skip-beginning-hd-get-all-levels-of-marked[OF LS', of M]
        get-level-skip-in-all-not-marked[of rev M L backtrack-lvl S']
      unfolding lits-of-def btS M
      by auto
    qed
  qed
}
ultimately have conflict-is-false-with-level S'' by blast
}
moreover
{
  assume conflicting S'  $\neq$  C-True
  have no-clause-is-false S' using  $\langle \text{conflicting } S' \neq C\text{-True} \rangle$  by auto
  then have conflict-is-false-with-level S'' using calculation(3) by blast
}
ultimately show ?case by fast
qed

lemma rtranclp-cdclW-stgy-no-smaller-confl-inv:
  assumes
    cdclW-stgy** S S' and
    n-l: no-smaller-confl S and
    cls-false: conflict-is-false-with-level S and
    lev: cdclW-M-level-inv S and
    no-f: no-clause-is-false S and
    dist: distinct-cdclW-state S and
    conflicting: cdclW-conflicting S and
    decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
    learned: cdclW-learned-clause S and
    alien: no-strange-atm S
  shows no-smaller-confl S'  $\wedge$  conflict-is-false-with-level S'
  using assms(1)
proof (induct rule: rtranclp-induct)
  case base
  then show ?case using n-l cls-false by auto
next
  case (step S' S'') note st = this(1) and cdcl = this(2) and IH = this(3)
  have no-smaller-confl S' and conflict-is-false-with-level S'
    using IH by blast+
  moreover have cdclW-M-level-inv S'
    using st lev rtranclp-cdclW-stgy-rtranclp-cdclW
    by (blast intro: rtranclp-cdclW-consistent-inv)+
  moreover have no-clause-is-false S'
    using st no-f rtranclp-cdclW-stgy-not-non-negated-init-clss by blast
  moreover have distinct-cdclW-state S'
    using rtranclp-distinct-cdclW-state-inv[of S S'] lev rtranclp-cdclW-stgy-rtranclp-cdclW[OF st]
    dist by auto
  moreover have cdclW-conflicting S'
    using rtranclp-cdclW-all-inv(6)[of S S'] st alien conflicting decomp dist learned lev
    rtranclp-cdclW-stgy-rtranclp-cdclW by blast
  ultimately show ?case

```

using *cdcl<sub>W</sub>-stgy-no-smaller-confl*[*OF cdcl*] *cdcl<sub>W</sub>-stgy-ex-lit-of-max-level*[*OF cdcl*] by *fast*  
qed

### 17.6.7 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  $(\text{conflicting } S' = C\text{-Clause } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S')))$   
 $\vee (\text{conflicting } S' = C\text{-True} \wedge \text{trail } S' \models_{\text{asm}} \text{init-clss } S')$

**proof** –

let *?S* = *init-state N*

**have**

*termi*:  $\forall S''. \neg \text{cdcl}_W\text{-stgy } S' S''$  **and**

*step*: *cdcl<sub>W</sub>-stgy*<sup>\*</sup> (*init-state N*) *S'* **using** *full unfolding full-def* **by** *auto*

**moreover have**

*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**

*confl*: *cdcl<sub>W</sub>-conflicting S'* **and**

*decomp*: *all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))*

**using** *no-d tranclp-cdcl<sub>W</sub>-stgy-tranclp-cdcl<sub>W</sub>*[*of ?S S'*] *step rtranclp-cdcl<sub>W</sub>-all-inv(1-6)*[*of ?S S'*]

**unfolding** *rtranclp-unfold* **by** *auto*

**moreover**

**have**  $\forall D \in \#N. \neg [] \models_{\text{as}} C\text{Not } D$  **using** *no-empty* **by** *auto*

**then have** *confl-k*: *conflict-is-false-with-level S'*

**using** *rtranclp-cdcl<sub>W</sub>-stgy-no-smaller-confl-inv*[*OF step*] *no-d* **by** *auto*

**show** *?thesis*

**using** *cdcl<sub>W</sub>-stgy-final-state-conclusive*[*OF termi decomp learned level-inv alien no-dup confl confl-k*] .

qed

**lemma** *conflict-is-full1-cdcl<sub>W</sub>-cp*:

assumes *cp*: *conflict S S'*

shows *full1 cdcl<sub>W</sub>-cp S S'*

**proof** –

**have** *cdcl<sub>W</sub>-cp S S'* **and** *conflicting S' ≠ C-True* **using** *cp cdcl<sub>W</sub>-cp.intros* **by** *auto*

**then have** *cdcl<sub>W</sub>-cp*<sup>++</sup> *S S'* **by** *blast*

**moreover have** *no-step cdcl<sub>W</sub>-cp S'*

**using**  $(\text{conflicting } S' \neq C\text{-True})$  **by**  $(\text{metis } \text{cdcl}_W\text{-cp-conflicting-not-empty } \text{conflicting-clause.exhaust})$

**ultimately show** *full1 cdcl<sub>W</sub>-cp S S'* **unfolding** *full1-def* **by** *blast+*

qed

**lemma** *cdcl<sub>W</sub>-cp-fst-empty-conflicting-false*:

assumes *cdcl<sub>W</sub>-cp S S'*

and *trail S* = []

and *conflicting S ≠ C-True*

shows *False*

**using** *assms* **by**  $(\text{induct rule: } \text{cdcl}_W\text{-cp.induct})$  *auto*

**lemma** *cdcl<sub>W</sub>-o-fst-empty-conflicting-false*:

```

assumes cdclW-o S S'
and trail S = []
and conflicting S ≠ C-True
shows False
using assms by (induct rule: cdclW-o-induct) auto

lemma cdclW-stgy-fst-empty-conflicting-false:
  assumes cdclW-stgy S S'
  and trail S = []
  and conflicting S ≠ C-True
  shows False
  using assms apply (induct rule: cdclW-stgy.induct)
  using trancpD cdclW-cp-fst-empty-conflicting-false unfolding full1-def apply metis
  using cdclW-o-fst-empty-conflicting-false by blast
thm cdclW-cp.induct[split-format(complete)]

lemma cdclW-cp-conflicting-is-false:
  cdclW-cp S S' ⇒ conflicting S = C-Clause {#} ⇒ False
  by (induction rule: cdclW-cp.induct) auto

lemma rtrancp-cdclW-cp-conflicting-is-false:
  cdclW-cp++ S S' ⇒ conflicting S = C-Clause {#} ⇒ False
  apply (induction rule: trancp.induct)
  by (auto dest: cdclW-cp-conflicting-is-false)

lemma cdclW-o-conflicting-is-false:
  cdclW-o S S' ⇒ conflicting S = C-Clause {#} ⇒ False
  by (induction rule: cdclW-o-induct) auto

lemma cdclW-stgy-conflicting-is-false:
  cdclW-stgy S S' ⇒ conflicting S = C-Clause {#} ⇒ False
  apply (induction rule: cdclW-stgy.induct)
  unfolding full1-def apply (metis (no-types) cdclW-cp-conflicting-not-empty trancpD)
  unfolding full-def by (metis conflict-with-false-implies-terminated other)

lemma rtrancp-cdclW-stgy-conflicting-is-false:
  cdclW-stgy** S S' ⇒ conflicting S = C-Clause {#} ⇒ S' = S
  apply (induction rule: rtrancp-induct)
  apply simp
  using cdclW-stgy-conflicting-is-false by blast

lemma full-cdclW-init-clss-with-false-normal-form:
  assumes
     $\forall m \in \text{set } M. \neg \text{is-marked } m$  and
    E = C-Clause D and
    state S = (M, N, U, 0, E)
    full cdclW-stgy S S' and
    all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))
    cdclW-learned-clause S
    cdclW-M-level-inv S
    no-strange-atm S
    distinct-cdclW-state S
    cdclW-conflicting S
  shows  $\exists M''. \text{state } S' = (M'', N, U, 0, \text{C-Clause } \{ \# \})$ 

```



```

using assms(10,9,8,7,6,5,4,3,2,1)
proof (induction M arbitrary: E D S)
  case Nil
  then show ?case
    using rtrancp-cdclW-stgy-conflicting-is-false unfolding full-def cdclW-conflicting-def by auto
next
case (Cons L M) note IH = this(1) and full = this(8) and E = this(10) and inv = this(2-7) and
  S = this(9) and nm = this(11)
obtain K p where K: L = Propagated K p
  using nm by (cases L) auto
have every-mark-is-a-conflict S using inv unfolding cdclW-conflicting-def by auto
then have MpK: M ⊨as CNot ( p - {#K#}) and Kp: K ∈# p
  using S unfolding K by fastforce+
then have p: p = ( p - {#K#}) + {#K#}
  by (auto simp add: multiset-eq-iff)
then have K': L = Propagated K ( (( p - {#K#}) + {#K#}))
  using K by auto

consider (D) D = {#} | (D') D ≠ {#} by blast
then show ?case
  proof cases
    case D
    then show ?thesis
      using full rtrancp-cdclW-stgy-conflicting-is-false S unfolding full-def E D by auto
  next
    case D'
    then have no-p: no-step propagate S and no-c: no-step conflict S
      using S E by auto
    then have no-step cdclW-cp S by (auto simp: cdclW-cp.simps)
    have res-skip: ∃ T. (resolve S T ∧ no-step skip S ∧ full cdclW-cp T T)
       $\vee$  (skip S T ∧ no-step resolve S ∧ full cdclW-cp T T)
    proof cases
      assume  $\neg \text{lit-of } L \notin \# D$ 
      then obtain T where sk: skip S T and res: no-step resolve S
      using S that D' K unfolding skip.simps E by fastforce
      have full cdclW-cp T T
      using sk by (auto simp add: conflicting-clause-full-cdclW-cp)
      then show ?thesis
      using sk res by blast
    next
      assume LD: ¬lit-of L ∉# D
      then have D: C-Clause D = C-Clause ((D - {#lit-of L#}) + {#lit-of L#})
      by (auto simp add: multiset-eq-iff)

    have  $\bigwedge L. \text{get-level } L \ M = 0$ 
      by (simp add: nm)
    then have get-maximum-level (D - {#lit-of L#})
      (Propagated K ( ( p - {#K#}) + {#K#})) # M) = 0
      using LD get-maximum-level-exists-lit-of-max-level
    proof -
      obtain L' where get-level L' (L#M) = get-maximum-level D (L#M)
      using LD get-maximum-level-exists-lit-of-max-level[of D L#M] by fastforce
      then show ?thesis by (metis (mono-tags) K' bex-msetE get-level-skip-all-not-marked
        get-maximum-level-exists-lit nm not-gr0)
    qed

```

```

then obtain  $T$  where  $sk$ : resolve  $S$   $T$  and  $res$ : no-step skip  $S$ 
  using resolve-rule[of  $S$   $K$   $p - \{\#K\# \} M N U 0 (D - \{\#-K\# \})$ 
    update-conflicting (C-Clause (remdups-mset ( $D - \{\#-K\# \} + (p - \{\#K\# \})$ )))] (tl-trail  $S$ )]
   $S$  unfolding  $K' D E$  by fastforce
have full cdclW-cp  $T$   $T$ 
  using  $sk$  by (auto simp add: conflicting-clause-full-cdclW-cp)
then show ?thesis
  using  $sk$   $res$  by blast
qed
then have step-s:  $\exists T. \text{cdcl}_W\text{-stgy } S T$ 
  using (no-step cdclW-cp  $S$ ) other' by (meson bj resolve skip)
have get-all-marked-decomposition ( $L \# M$ ) =  $[(\square, L\#M)]$ 
  using nm unfolding  $K$  apply (induction  $M$  rule: marked-lit-list-induct, simp)
  by (case-tac hd (get-all-marked-decomposition  $xs$ ), auto)+
then have no-b: no-step backtrack  $S$ 
  using nm  $S$  by auto
have no-d: no-step decide  $S$ 
  using  $S$   $E$  by auto

have full-S-S: full cdclW-cp  $S$   $S$ 
  using  $S$   $E$  by (auto simp add: conflicting-clause-full-cdclW-cp)
then have no-f: no-step (full1 cdclW-cp)  $S$ 
  unfolding full-def full1-def rtrancp-unfold by (meson trancpD)
obtain  $T$  where
   $s$ : cdclW-stgy  $S$   $T$  and  $st$ : cdclW-stgy**  $T$   $S'$ 
  using full step-s full unfolding full-def by (metis rtrancp-unfold trancpD)
have resolve  $S$   $T \vee \text{skip } S$   $T$ 
  using  $s$  no-b no-d res-skip full-S-S unfolding cdclW-stgy.simps cdclW-o.simps full-unfold full1-def
  by (auto dest!: trancpD simp: cdclW-bj.simps)
then obtain  $D'$  where  $T$ : state  $T = (M, N, U, 0, \text{C-Clause } D')$ 
  using  $S$   $E$  by auto

have st-c: cdclW**  $S$   $T$ 
  using  $E$   $T$  rtrancp-cdclW-stgy-rtrancp-cdclW  $s$  by blast
have cdclW-conflicting  $T$ 
  using rtrancp-cdclW-all-inv(6)[OF st-c inv(6,5,4,3,2,1)] .
show ?thesis
  apply (rule IH[of  $T$ ])
    using rtrancp-cdclW-all-inv(6)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancp-cdclW-all-inv(5)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancp-cdclW-all-inv(4)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancp-cdclW-all-inv(3)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancp-cdclW-all-inv(2)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancp-cdclW-all-inv(1)[OF st-c inv(6,5,4,3,2,1)] apply blast
  apply (metis full-def st full)
  using  $T$   $E$  apply blast
  apply auto[]
  using nm by simp
qed
qed

lemma full-cdclW-stgy-final-state-conclusive-is-one-false:
  fixes  $S' :: 'st$ 
  assumes full: full cdclW-stgy (init-state  $N$ )  $S'$ 

```

**and** *no-d*: *distinct-mset-mset*  $N$   
**and** *empty*:  $\{\#\} \in \# \ N$   
**shows** *conflicting*  $S' = C\text{-Clause } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S'))$   
**proof** –  
**let**  $?S = \text{init-state } N$   
**have**  $\text{cdcl}_W\text{-stgy}^{**} ?S \ S'$  **and** *no-step*  $\text{cdcl}_W\text{-stgy } S'$  **using** *full unfolding full-def* **by** *auto*  
**then** **have** *plus-or-eq*:  $\text{cdcl}_W\text{-stgy}^{++} ?S \ S' \vee S' = ?S$  **unfolding** *rtranclp-unfold* **by** *auto*  
**have**  $\exists S''.$  *conflict*  $?S \ S''$  **using** *empty not-conflict-not-any-negated-init-clss* **by** *force*  
  
**then** **have**  $\text{cdcl}_W\text{-stgy}: \exists S'. \text{cdcl}_W\text{-stgy } ?S \ S'$   
**using**  $\text{cdcl}_W\text{-cp.conflict}'[\text{of } ?S]$  *conflict-is-full1-cdcl<sub>W</sub>-cp*  $\text{cdcl}_W\text{-stgy.intros}(1)$  **by** *metis*  
**have**  $S' \neq ?S$  **using** *(no-step cdcl<sub>W</sub>-stgy S')*  $\text{cdcl}_W\text{-stgy}$  **by** *blast*  
  
**then** **obtain**  $St:: 'st$  **where**  $St: \text{cdcl}_W\text{-stgy } ?S \ St$  **and**  $\text{cdcl}_W\text{-stgy}^{**} St \ S'$   
**using** *plus-or-eq* **by** *(metis (no-types) (cdcl<sub>W</sub>-stgy<sup>\*\*</sup> ?S S') converse-rtranclpE)*  
**have**  $st: \text{cdcl}_W^{**} ?S \ St$   
**by** *(simp add: rtranclp-unfold (cdcl<sub>W</sub>-stgy ?S St) cdcl<sub>W</sub>-stgy-tranclp-cdcl<sub>W</sub>)*  
  
**have**  $\exists T.$  *conflict*  $?S \ T$   
**using** *empty not-conflict-not-any-negated-init-clss* **by** *force*  
**then** **have**  $\text{fullSt}: \text{full1 } \text{cdcl}_W\text{-cp } ?S \ St$   
**using**  $St$  **unfolding**  $\text{cdcl}_W\text{-stgy.simps}$  **by** *blast*  
**then** **have**  $bt: \text{backtrack-lvl } St = (0::\text{nat})$   
**using**  $\text{rtranclp-cdcl}_W\text{-cp-backtrack-lvl}$  **unfolding** *full1-def*  
**by** *(fastforce dest!: tranclp-into-rtranclp)*  
**have**  $\text{cls-St}: \text{init-clss } St = N$   
**using**  $\text{fullSt } \text{cdcl}_W\text{-stgy-no-more-init-clss}[OF \ St]$  **by** *auto*  
**have** *conflicting*  $St \neq C\text{-True}$   
**proof** *(rule ccontr)*  
**assume**  $\neg ?thesis$   
**then** **have**  $\exists T.$  *conflict*  $St \ T$   
**using** *empty cls-St* **by** *(fastforce simp: clauses-def)*  
**then** **show** *False* **using**  $\text{fullSt}$  **unfolding** *full1-def* **by** *blast*  
**qed**  
  
**have**  $1: \forall m \in \text{set } (\text{trail } St). \neg \text{is-marked } m$   
**using**  $\text{fullSt}$  **unfolding** *full1-def* **by** *(auto dest!: tranclp-into-rtranclp rtranclp-cdcl<sub>W</sub>-cp-dropWhile-trail)*  
**have**  $2: \text{full } \text{cdcl}_W\text{-stgy } St \ S'$   
**using**  $(\text{cdcl}_W\text{-stgy}^{**} St \ S')$  *(no-step cdcl<sub>W</sub>-stgy S')*  $bt$  **unfolding** *full-def* **by** *auto*  
**have**  $3: \text{all-decomposition-implies-m}$   
 $(\text{init-clss } St)$   
 $(\text{get-all-marked-decomposition}$   
 $(\text{trail } St))$   
**using**  $\text{rtranclp-cdcl}_W\text{-all-inv}(1)[OF \ st]$  *no-d bt* **by** *simp*  
**have**  $4: \text{cdcl}_W\text{-learned-clause } St$   
**using**  $\text{rtranclp-cdcl}_W\text{-all-inv}(2)[OF \ st]$  *no-d bt bt* **by** *simp*  
**have**  $5: \text{cdcl}_W\text{-M-level-inv } St$   
**using**  $\text{rtranclp-cdcl}_W\text{-all-inv}(3)[OF \ st]$  *no-d bt* **by** *simp*  
**have**  $6: \text{no-strange-atm } St$   
**using**  $\text{rtranclp-cdcl}_W\text{-all-inv}(4)[OF \ st]$  *no-d bt* **by** *simp*  
**have**  $7: \text{distinct-cdcl}_W\text{-state } St$   
**using**  $\text{rtranclp-cdcl}_W\text{-all-inv}(5)[OF \ st]$  *no-d bt* **by** *simp*  
**have**  $8: \text{cdcl}_W\text{-conflicting } St$   
**using**  $\text{rtranclp-cdcl}_W\text{-all-inv}(6)[OF \ st]$  *no-d bt* **by** *simp*

```

have init-clss S' = init-clss St and conflicting S' = C-Clause {#}
  using ⟨conflicting St ≠ C-True⟩ full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St]
  2 3 4 5 6 7 8 St apply (metis ⟨cdclW-stgy** St S'⟩ rtranclp-cdclW-stgy-no-more-init-clss)
  using ⟨conflicting St ≠ C-True⟩ full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St - -
    S'] 2 3 4 5 6 7 8 by (metis bt conflicting-clause.exhaust prod.inject)

moreover have init-clss S' = N
  using ⟨cdclW-stgy** (init-state N) S'⟩ rtranclp-cdclW-stgy-no-more-init-clss by fastforce
moreover have unsatisfiable (set-mset N)
  by (meson empty mem-set-mset-iff satisfiable-def true-clss-empty true-clss-def)
ultimately show ?thesis by auto
qed

```

**lemma** *full-cdcl<sub>W</sub>-stgy-final-state-conclusive*:

```

fixes S' :: 'st
assumes full: full cdclW-stgy (init-state N) S' and no-d: distinct-mset-mset N
shows (conflicting S' = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss S')))
  ∨ (conflicting S' = C-True ∧ trail S' ⊨asm init-clss S')
using assms full-cdclW-stgy-final-state-conclusive-is-one-false
full-cdclW-stgy-final-state-conclusive-non-false by blast

```

**lemma** *full-cdcl<sub>W</sub>-stgy-final-state-conclusive-from-init-state*:

```

fixes S' :: 'st
assumes full: full cdclW-stgy (init-state N) S'
and no-d: distinct-mset-mset N
shows (conflicting S' = C-Clause {#} ∧ unsatisfiable (set-mset N))
  ∨ (conflicting S' = C-True ∧ trail S' ⊨asm N ∧ satisfiable (set-mset N))

```

**proof** –

```

have N: init-clss S' = N
  using full unfolding full-def by (auto dest: rtranclp-cdclW-stgy-no-more-init-clss)
consider
  (confl) conflicting S' = C-Clause {#} and unsatisfiable (set-mset (init-clss S'))
  | (sat) conflicting S' = C-True and trail S' ⊨asm init-clss S'
  using full-cdclW-stgy-final-state-conclusive[OF assms] by auto
then show ?thesis
  proof cases
    case confl
    then show ?thesis by (auto simp: N)
  next
    case sat
    have cdclW-M-level-inv (init-state N) by auto
    then have cdclW-M-level-inv S'
      using full rtranclp-cdclW-stgy-consistent-inv unfolding full-def by blast
    then have consistent-interp (lits-of (trail S')) unfolding cdclW-M-level-inv-def by blast
    moreover have lits-of (trail S') ⊨s set-mset (init-clss S')
      using sat(2) by (auto simp add: true-annot-def true-annot-def true-clss-def)
    ultimately have satisfiable (set-mset (init-clss S')) by simp
    then show ?thesis using sat unfolding N by blast
  qed

```

**qed**

**qed**

**end**

**end**

**theory** *CDCL-W-Termination*

**imports** *CDCL-W*

**begin**

**context** *cdcl<sub>W</sub>-ops*

**begin**

## 17.7 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 *build-all-simple-clss*.

The invariant contains all the structural invariants that holds,

**definition** *cdcl<sub>W</sub>-all-struct-inv* **where**

*cdcl<sub>W</sub>-all-struct-inv* *S* =  
 (no-strange-atm *S*  $\wedge$  *cdcl<sub>W</sub>-M-level-inv* *S*  
 $\wedge$  ( $\forall s \in \#$  learned-clss *S*.  $\neg$ tautology *s*)  
 $\wedge$  distinct-cdcl<sub>W</sub>-state *S*  $\wedge$  *cdcl<sub>W</sub>-conflicting* *S*  
 $\wedge$  all-decomposition-implies-m (init-clss *S*) (get-all-marked-decomposition (trail *S*))  
 $\wedge$  *cdcl<sub>W</sub>-learned-clause* *S*)

**lemma** *cdcl<sub>W</sub>-all-struct-inv-inv*:

**assumes** *cdcl<sub>W</sub>* *S* *S'* **and** *cdcl<sub>W</sub>-all-struct-inv* *S*

**shows** *cdcl<sub>W</sub>-all-struct-inv* *S'*

**unfolding** *cdcl<sub>W</sub>-all-struct-inv-def*

**proof** (intro *HOL.conjI*)

**show** no-strange-atm *S'*

**using** *cdcl<sub>W</sub>-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *auto*

**show** *cdcl<sub>W</sub>-M-level-inv* *S'*

**using** *cdcl<sub>W</sub>-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *fast*

**show** distinct-cdcl<sub>W</sub>-state *S'*

**using** *cdcl<sub>W</sub>-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *fast*

**show** *cdcl<sub>W</sub>-conflicting* *S'*

**using** *cdcl<sub>W</sub>-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *fast*

**show** all-decomposition-implies-m (init-clss *S'*) (get-all-marked-decomposition (trail *S'*))

**using** *cdcl<sub>W</sub>-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *fast*

**show** *cdcl<sub>W</sub>-learned-clause* *S'*

**using** *cdcl<sub>W</sub>-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *fast*

**show**  $\forall s \in \#$ learned-clss *S'*.  $\neg$  tautology *s*

**using** *assms*(1)[*THEN* learned-clss-are-not-tautologies] *assms*(2)

**unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *fast*

**qed**

**lemma** *rtrancpl-cdcl<sub>W</sub>-all-struct-inv-inv*:

**assumes** *cdcl<sub>W</sub>\*\** *S* *S'* **and** *cdcl<sub>W</sub>-all-struct-inv* *S*

**shows** *cdcl<sub>W</sub>-all-struct-inv* *S'*

**using** *assms* **by** induction (auto intro: *cdcl<sub>W</sub>-all-struct-inv-inv*)

**lemma** *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-all-struct-inv*:

*cdcl<sub>W</sub>-stgy* *S* *T*  $\implies$  *cdcl<sub>W</sub>-all-struct-inv* *S*  $\implies$  *cdcl<sub>W</sub>-all-struct-inv* *T*

**by** (meson *cdcl<sub>W</sub>-stgy-trancpl-cdcl<sub>W</sub>* *rtrancpl-cdcl<sub>W</sub>-all-struct-inv-inv* *rtrancpl-unfold*)

**lemma** *rtrancpl-cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-all-struct-inv*:

*cdcl<sub>W</sub>-stgy\*\** *S* *T*  $\implies$  *cdcl<sub>W</sub>-all-struct-inv* *S*  $\implies$  *cdcl<sub>W</sub>-all-struct-inv* *T*

**by** (induction rule: *rtrancpl-induct*) (auto intro: *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-all-struct-inv*)

## 17.8 No Relearning of a clause

This is only proved in simple cases by auto. In assumptions, nothing happens. and  $?P$  (*if*  $?Q$  then  $?x$  else  $?y$ ) =  $(\neg (?Q \wedge \neg ?P ?x \vee \neg ?Q \wedge \neg ?P ?y))$  can blow up goals.

**lemma** *if-0-1-ge-0*:

$0 < (\text{if } P \text{ then } a \text{ else } (0::\text{nat})) \longleftrightarrow P \wedge 0 < a$   
by auto

**lemma** *cdcl<sub>W</sub>-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<sub>W</sub>*: *cdcl<sub>W</sub>-o*  $S \ T$  **and**  
*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows** *backtrack*  $S \ T \wedge \text{conflicting } S = C\text{-Clause } D$   
**using** *cdcl<sub>W</sub>* *lev* *learned* *new*  
**proof** (*induction rule*: *cdcl<sub>W</sub>-o-induct-lev2*)  
**case** (*backtrack*  $K \ i \ M1 \ M2 \ L \ C \ T$ ) **note** *decomp* = *this*(1) **and** *undef* = *this*(6) **and**  $T = \text{this}(7)$   
**and**  
 $D \cdot T = \text{this}(9)$  **and**  $D \cdot S = \text{this}(10)$   
**then have**  $D = C + \{\#L\# \}$   
**using** *not-gr0* *lev* **by** (*auto simp*: *cdcl<sub>W</sub>-M-level-inv-decomp if-0-1-ge-0*)  
**then show** *?case*  
**using**  $T$  *backtrack.hyps*(1–5) *backtrack.intros* **by** auto  
**qed** auto

**lemma** *cdcl<sub>W</sub>-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<sub>W</sub>*: *cdcl<sub>W</sub>-stgy*  $S \ T$  **and**  
*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows**  $\exists S'. \text{backtrack } S \ S' \wedge \text{cdcl}_W\text{-stgy}^{**} \ S' \ T \wedge \text{conflicting } S = C\text{-Clause } D$   
**using** *cdcl<sub>W</sub>* *learned* *new*  
**proof** (*induction rule*: *cdcl<sub>W</sub>-stgy.induct*)  
**case** (*conflict'*  $S'$ )  
**then show** *?case*  
**unfolding** *full1-def* **by** (*metis* (*mono-tags*, *lifting*) *rtranclp-cdcl<sub>W</sub>-cp-learned-clause-inv*  
*trancplp-into-rtranclp*)  
**next**  
**case** (*other'*  $S' \ S''$ )  
**then have**  $D \in \# \text{ learned-clss } S'$   
**unfolding** *full-def* **by** (*auto dest*: *rtranclp-cdcl<sub>W</sub>-cp-learned-clause-inv*)  
**then show** *?case*  
**using** *cdcl<sub>W</sub>-o-new-clause-learned-is-backtrack-step*[*OF* -  $\langle D \notin \# \text{ learned-clss } S \rangle \langle \text{cdcl}_W\text{-o } S \ S' \rangle$ ]  
 $\langle \text{full } \text{cdcl}_W\text{-cp } S' \ S'' \rangle$  *lev* **by** (*metis* *cdcl<sub>W</sub>-stgy.conflict'* *full-unfold r-into-rtranclp*  
*rtranclp.rtrancl-refl*)  
**qed**

**lemma** *rtranclp-cdcl<sub>W</sub>-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<sub>W</sub>*: *cdcl<sub>W</sub>-stgy*<sup>\*\*</sup>  $S \ T$  **and**  
*lev*: *cdcl<sub>W</sub>-M-level-inv*  $S$   
**shows**  $\exists S' \ S''. \text{cdcl}_W\text{-stgy}^{**} \ S \ S' \wedge \text{backtrack } S' \ S'' \wedge \text{conflicting } S' = C\text{-Clause } D \wedge$   
 $\text{cdcl}_W\text{-stgy}^{**} \ S'' \ T$   
**using** *cdcl<sub>W</sub>* *learned* *new*

```

proof (induction rule: rtrancpl-induct)
  case base
  then show ?case by blast
next
  case (step T U) note st = this(1) and o = this(2) and IH = this(3) and
    D-U = this(4) and D-S = this(5)
  show ?case
  proof (cases D ∈# learned-clss T)
    case True
    then obtain S' S'' where
      st': cdclW-stgy** S S' and
      bt: backtrack S' S'' and
      confl: conflicting S' = C-Clause D and
      st'': cdclW-stgy** S'' T
    using IH D-S by metis
    then show ?thesis using o by (meson rtrancpl.simps)
  next
  case False
  have cdclW-M-level-inv T
    using lev rtrancpl-cdclW-stgy-consistent-inv st by blast
  then obtain S' where
    bt: backtrack T S' and
    st': cdclW-stgy** S' U and
    confl: conflicting T = C-Clause D
  using cdclW-cp-new-clause-learned-has-backtrack-step[OF D-U False o]
    by metis
  then have cdclW-stgy** S T and
    backtrack T S' and
    conflicting T = C-Clause D and
    cdclW-stgy** S' U
  using o st by auto
  then show ?thesis by blast
qed
qed

```

```

lemma propagate-no-more-Marked-lit:
  assumes propagate S S'
  shows Marked K i ∈ set (trail S)  $\longleftrightarrow$  Marked K i ∈ set (trail S')
  using assms by auto

```

```

lemma conflict-no-more-Marked-lit:
  assumes conflict S S'
  shows Marked K i ∈ set (trail S)  $\longleftrightarrow$  Marked K i ∈ set (trail S')
  using assms by auto

```

```

lemma cdclW-cp-no-more-Marked-lit:
  assumes cdclW-cp S S'
  shows Marked K i ∈ set (trail S)  $\longleftrightarrow$  Marked K i ∈ set (trail S')
  using assms apply (induct rule: cdclW-cp.induct)
  using conflict-no-more-Marked-lit propagate-no-more-Marked-lit by auto

```

```

lemma rtrancpl-cdclW-cp-no-more-Marked-lit:
  assumes cdclW-cp** S S'
  shows Marked K i ∈ set (trail S)  $\longleftrightarrow$  Marked K i ∈ set (trail S')
  using assms apply (induct rule: rtrancpl-induct)

```

```

using cdclW-cp-no-more-Marked-lit by blast+

lemma cdclW-o-no-more-Marked-lit:
  assumes cdclW-o S S' and cdclW-M-level-inv S and  $\neg \text{decide } S S'$ 
  shows Marked K i  $\in$  set (trail S')  $\longrightarrow$  Marked K i  $\in$  set (trail S)
  using assms
proof (induct rule: cdclW-o-induct-lev2)
  case backtrack note decomp = this(1) and undef = this(6) and T = this(7) and lev = this(8)
  then show ?case
    by (auto simp: cdclW-M-level-inv-decomp)
next
  case (decide L T)
  then show ?case by blast
qed auto

lemma cdclW-new-marked-at-beginning-is-decide:
  assumes cdclW-stgy S S' and
  lev: cdclW-M-level-inv S and
  trail S' = M' @ Marked L i # M and
  trail S = M
  shows  $\exists T. \text{decide } S T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$ 
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S') note st = this(1) and no-dup = this(2) and S' = this(3) and S = this(4)
  have cdclW-M-level-inv S'
    using full1-cdclW-cp-consistent-inv no-dup st by blast
  then have Marked L i  $\in$  set (trail S') and Marked L i  $\notin$  set (trail S)
    using no-dup unfolding S S' cdclW-M-level-inv-def by (auto simp add: rev-image-eqI)
  then have False
    using st rtranclp-cdclW-cp-no-more-Marked-lit[of S S']
    unfolding full1-def rtranclp-unfold by blast
  then show ?case by fast
next
  case (other' T U) note o = this(1) and ns = this(2) and st = this(3) and no-dup = this(4) and
    S' = this(5) and S = this(6)
  have cdclW-M-level-inv U
    by (metis (full-types) lev cdclW.simps cdclW-consistent-inv full-def o
      other'.hyps(3) rtranclp-cdclW-cp-consistent-inv)
  then have Marked L i  $\in$  set (trail U) and Marked L i  $\notin$  set (trail S)
    using no-dup unfolding S S' cdclW-M-level-inv-def by (auto simp add: rev-image-eqI)
  then have Marked L i  $\in$  set (trail T)
    using st rtranclp-cdclW-cp-no-more-Marked-lit unfolding full-def by blast
  then show ?case
    using cdclW-o-no-more-Marked-lit[OF o]  $\langle$ Marked L i  $\notin$  set (trail S) $\rangle$  ns lev by meson
qed

lemma cdclW-o-is-decide:
  assumes cdclW-o S' T and cdclW-M-level-inv S'
  trail T = drop (length M0) M' @ Marked L i # H @ M and
   $\neg (\exists M'. \text{trail } S' = M' @ \text{Marked } L i \# H @ M)$ 
  shows decide S' T
  using assms
proof (induction rule: cdclW-o-induct-lev2)
  case (backtrack K i M1 M2 L D)
  then obtain c where trail S' = c @ M2 @ Marked K (Suc i) # M1

```



```

  by auto
then show ?case
  using backtrack by (cases drop (length M0) M') (auto simp: cdclW-M-level-inv-def)
next
  case decide
  show ?case using decide-rule[of S'] decide(1-4) by auto
qed auto

lemma rtrancpl-cdclW-new-marked-at-beginning-is-decide:
  assumes cdclW-stgy** R U and
  trail U = M' @ Marked L i # H @ M and
  trail R = M and
  cdclW-M-level-inv R
  shows
     $\exists S T T'. \text{cdcl}_W\text{-stgy}^{**} R S \wedge \text{decide } S T \wedge \text{cdcl}_W\text{-stgy}^{**} T U \wedge \text{cdcl}_W\text{-stgy}^{**} S U \wedge$ 
     $\text{no-step } \text{cdcl}_W\text{-cp } S \wedge \text{trail } T = \text{Marked } L i \# H @ M \wedge \text{trail } S = H @ M \wedge \text{cdcl}_W\text{-stgy } S T' \wedge$ 
     $\text{cdcl}_W\text{-stgy}^{**} T' U$ 
  using assms
proof (induct arbitrary: M H M' i rule: rtrancpl-induct)
  case base
  then show ?case by auto
next
  case (step T U) note st = this(1) and IH = this(3) and s = this(2) and
    U = this(4) and S = this(5) and lev = this(6)
  show ?case
  proof (cases  $\exists M'. \text{trail } T = M' @ \text{Marked } L i \# H @ M$ )
    case False
    with s show ?thesis using U s st S
  proof induction
    case (conflict' W) note cp = this(1) and nd = this(2) and W = this(3)
    then obtain M0 where trail W = M0 @ trail T and nmarked:  $\forall l \in \text{set } M_0. \neg \text{is-marked } l$ 
      using rtrancpl-cdclW-cp-dropWhile-trail unfolding full1-def rtrancpl-unfold by meson
    then have MV:  $M' @ \text{Marked } L i \# H @ M = M_0 @ \text{trail } T$  unfolding W by simp
    then have V: trail T = drop (length M0) (M' @ Marked L i # H @ M)
      by auto
    have takeWhile (Not o is-marked) M' = M0 @ takeWhile (Not o is-marked) (trail T)
      using arg-cong[OF MV, of takeWhile (Not o is-marked)] nmarked
      by (simp add: takeWhile-tail)
    from arg-cong[OF this, of length] have length M0 ≤ length M'
      unfolding length-append by (metis (no-types, lifting) Nat.le-trans le-add1
        length-takeWhile-le)
    then have False using nd V by auto
    then show ?case by fast
  next
    case (other' T' U) note o = this(1) and ns = this(2) and cp = this(3) and nd = this(4)
      and U = this(5) and st = this(6)
    obtain M0 where trail U = M0 @ trail T' and nmarked:  $\forall l \in \text{set } M_0. \neg \text{is-marked } l$ 
      using rtrancpl-cdclW-cp-dropWhile-trail cp unfolding full-def by meson
    then have MV:  $M' @ \text{Marked } L i \# H @ M = M_0 @ \text{trail } T'$  unfolding U by simp
    then have V: trail T' = drop (length M0) (M' @ Marked L i # H @ M)
      by auto
    have takeWhile (Not o is-marked) M' = M0 @ takeWhile (Not o is-marked) (trail T')
      using arg-cong[OF MV, of takeWhile (Not o is-marked)] nmarked
      by (simp add: takeWhile-tail)
    from arg-cong[OF this, of length] have length M0 ≤ length M'

```

```

    unfolding length-append by (metis (no-types, lifting) Nat.le-trans le-add1
      length-takeWhile-le)
  then have tr-T': trail T' = drop (length M0) M' @ Marked L i # H @ M using V by auto
  then have LT': Marked L i ∈ set (trail T') by auto
  moreover
    have cdclW-M-level-inv T
      using lev rtrancp-cdclW-stgy-consistent-inv step.hyps(1) by blast
    then have decide T T' using o nd tr-T' cdclW-o-is-decide by metis
  ultimately have decide T T' using cdclW-o-no-more-Marked-lit[OF o] by blast
  then have 1: cdclW-stgy** R T and 2: decide T T' and 3: cdclW-stgy** T' U
    using st other'.prems(4)
    by (metis cdclW-stgy.conflict' cp full-unfold r-into-rtrancp rtrancp.rtrancp-refl)+
  have [simp]: drop (length M0) M' = []
    using ⟨decide T T'⟩ ⟨Marked L i ∈ set (trail T')⟩ nd tr-T'
    by (auto simp add: Cons-eq-append-conv)
  have T': drop (length M0) M' @ Marked L i # H @ M = Marked L i # trail T
    using ⟨decide T T'⟩ ⟨Marked L i ∈ set (trail T')⟩ nd tr-T'
    by auto
  have trail T' = Marked L i # trail T
    using ⟨decide T T'⟩ ⟨Marked L i ∈ set (trail T')⟩ tr-T'
    by auto
  then have 5: trail T' = Marked L i # H @ M
    using append.simps(1) list.sel(3) local.other'(5) tl-append2 by (simp add: tr-T')
  have 6: trail T = H @ M
    by (metis (no-types) ⟨trail T' = Marked L i # trail T⟩
      ⟨trail T' = drop (length M0) M' @ Marked L i # H @ M⟩ append-Nil list.sel(3) nd
      tl-append2)
  have 7: cdclW-stgy** T U using other'.prems(4) st by auto
  have 8: cdclW-stgy T U cdclW-stgy** U U
    using cdclW-stgy.other'[OF other'(1-3)] by simp-all
  show ?case apply (rule exI[of - T], rule exI[of - T'], rule exI[of - U])
    using ns 1 2 3 5 6 7 8 by fast
qed
next
case True
then obtain M' where T: trail T = M' @ Marked L i # H @ M by metis
from IH[OF this S lev] obtain S' S'' S''' where
  1: cdclW-stgy** R S' and
  2: decide S' S'' and
  3: cdclW-stgy** S'' T and
  4: no-step cdclW-cp S' and
  6: trail S'' = Marked L i # H @ M and
  7: trail S' = H @ M and
  8: cdclW-stgy** S' T and
  9: cdclW-stgy S' S''' and
  10: cdclW-stgy** S''' T
  by blast
have cdclW-stgy** S'' U using s ⟨cdclW-stgy** S'' T⟩ by auto
moreover have cdclW-stgy** S' U using 8 s by auto
moreover have cdclW-stgy** S''' U using 10 s by auto
ultimately show ?thesis apply - apply (rule exI[of - S'], rule exI[of - S''])
  using 1 2 4 6 7 8 9 by blast
qed
qed

```

**lemma** *rtrancp-cdcl<sub>W</sub>-new-marked-at-beginning-is-decide'*:  
**assumes** *cdcl<sub>W</sub>-stgy\*\* R U* **and**  
*trail U = M' @ Marked L i # H @ M* **and**  
*trail R = M* **and**  
*cdcl<sub>W</sub>-M-level-inv R*  
**shows**  $\exists y y'. \text{cdcl}_W\text{-stgy}^{**} R y \wedge \text{cdcl}_W\text{-stgy } y y' \wedge \neg (\exists c. \text{trail } y = c @ \text{Marked } L i \# H @ M)$   
 $\wedge (\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L i \# H @ M))^{**} y' U$

**proof** –

**fix** *T'*

**obtain** *S' T T'* **where**

*st*: *cdcl<sub>W</sub>-stgy\*\* R S'* **and**

*decide S' T* **and**

*TU*: *cdcl<sub>W</sub>-stgy\*\* T U* **and**

*no-step cdcl<sub>W</sub>-cp S'* **and**

*trT*: *trail T = Marked L i # H @ M* **and**

*trS'*: *trail S' = H @ M* **and**

*S'U*: *cdcl<sub>W</sub>-stgy\*\* S' U* **and**

*S'T'*: *cdcl<sub>W</sub>-stgy S' T'* **and**

*T'U*: *cdcl<sub>W</sub>-stgy\*\* T' U*

**using** *rtrancp-cdcl<sub>W</sub>-new-marked-at-beginning-is-decide*[*OF assms*] **by** *blast*

**have** *n*:  $\neg (\exists c. \text{trail } S' = c @ \text{Marked } L i \# H @ M)$  **using** *trS'* **by** *auto*

**show** *?thesis*

**using** *rtrancp-trans*[*OF st*] *rtrancp-exists-last-with-prop*[*of cdcl<sub>W</sub>-stgy S' T' -*

*λa -. ¬(∃ c. trail a = c @ Marked L i # H @ M), OF S'T' T'U n*]

**by** *meson*

**qed**

**lemma** *beginning-not-marked-invert*:

**assumes** *A: M @ A = M' @ Marked K i # H* **and**

*nm*:  $\forall m \in \text{set } M. \neg \text{is-marked } m$

**shows**  $\exists M. A = M @ \text{Marked } K i \# H$

**proof** –

**have** *A = drop (length M) (M' @ Marked K i # H)*

**using** *arg-cong*[*OF A, of drop (length M)*] **by** *auto*

**moreover have** *drop (length M) (M' @ Marked K i # H) = drop (length M) M' @ Marked K i # H*

**using** *nm* **by** (*metis* (*no-types, lifting*) *A drop-Cons' drop-append marked-lit.disc(1) not-gr0*

*nth-append nth-append-length nth-mem zero-less-diff*)

**finally show** *?thesis* **by** *fast*

**qed**

**lemma** *cdcl<sub>W</sub>-stgy-trail-has-new-marked-is-decide-step*:

**assumes** *cdcl<sub>W</sub>-stgy S T*

$\neg (\exists c. \text{trail } S = c @ \text{Marked } L i \# H @ M)$  **and**

$(\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L i \# H @ M))^{**} T U$  **and**

$\exists M'. \text{trail } U = M' @ \text{Marked } L i \# H @ M$  **and**

*lev*: *cdcl<sub>W</sub>-M-level-inv S*

**shows**  $\exists S'. \text{decide } S S' \wedge \text{full } \text{cdcl}_W\text{-cp } S' T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$

**using** *assms*(3,1,2,4,5)

**proof** *induction*

**case** (*step T U*)

**then show** *?case* **by** *fastforce*

**next**

**case** *base*

**then show** *?case*

**proof** (*induction rule: cdcl<sub>W</sub>-stgy.induct*)

```

case (conflict' T) note cp = this(1) and nd = this(2) and M' = this(3) and no-dup = this(3)
then obtain M' where M': trail T = M' @ Marked L i # H @ M by metis
obtain M'' where M'': trail T = M'' @ trail S and nm:  $\forall m \in \text{set } M''. \neg \text{is-marked } m$ 
  using cp unfolding full1-def
  by (metis rtranclp-cdclW-cp-dropWhile-trail' trancplp-into-rtranclp)
have False
  using beginning-not-marked-invert[of M'' trail S M' L i H @ M] M' nm nd unfolding M''
  by fast
then show ?case by fast
next
case (other' T U') note o = this(1) and ns = this(2) and cp = this(3) and nd = this(4)
  and trU' = this(5)
have cdclW-cp** T U' using cp unfolding full-def by blast
from rtranclp-cdclW-cp-dropWhile-trail[OF this]
have  $\exists M'. \text{trail } T = M' @ \text{Marked } L \ i \ \# \ H @ M$ 
  using trU' beginning-not-marked-invert[of - trail T - L i H @ M] by metis
then obtain M' where M': trail T = M' @ Marked L i # H @ M
  by auto
with o lev nd cp ns
show ?case
  proof (induction rule: cdclW-o-induct-lev2)
    case (decide L) note dec = this(1) and cp = this(5) and ns = this(4)
    then have decide S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))
      using decide.hyps decide.intros[of S] by force
    then show ?case using cp decide.premis by (meson decide-state-eq-compatible ns state-eq-ref
      state-eq-sym)
    next
    case (backtrack K j M1 M2 L' D T) note decomp = this(1) and cp = this(3)
      and undef = this(6) and T = this(7) and trT = this(12) and ns = this(4)
    obtain MS3 where MS3: trail S = MS3 @ M2 @ Marked K (Suc j) # M1
      using get-all-marked-decomposition-exists-prepend[OF decomp] by metis
    have tl (M' @ Marked L i # H @ M) = tl M' @ Marked L i # H @ M
      using lev trT T lev undef decomp by (cases M') (auto simp: cdclW-M-level-inv-decomp)
    then have M'': M1 = tl M' @ Marked L i # H @ M
      using arg-cong[OF trT[simplified], of tl] T decomp undef lev
      by (simp add: cdclW-M-level-inv-decomp)
    have False using nd MS3 T undef decomp unfolding M'' by auto
    then show ?case by fast
  qed auto
qed
qed

```

**lemma** *rtranclp-cdcl<sub>W</sub>-stgy-with-trail-end-has-trail-end*:

**assumes** ( $\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M)$ )\*\* *T* *U* **and**  
 $\exists M'. \text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M$   
**shows**  $\exists M'. \text{trail } T = M' @ \text{Marked } L \ i \ \# \ H @ M$   
**using** *assms* **by** (*induction rule: rtranclp-induct*) *auto*

**lemma** *cdcl<sub>W</sub>-o-cannot-learn*:

**assumes**  
*cdcl<sub>W</sub>-o* *y* *z* **and**  
*lev: cdcl<sub>W</sub>-M-level-inv* *y* **and**  
*trM: trail* *y* = *c* @ *Marked* *Kh* *i* # *H* **and**  
*DL: D* + {*#L#*}  $\notin$  *learned-clss* *y* **and**  
*DH: atms-of* *D*  $\subseteq$  *atm-of* *lits-of* *H* **and**

*LH*: *atm-of* *L*  $\notin$  *atm-of* 'lits-of *H* and  
*learned*:  $\forall T. \text{conflicting } y = C\text{-Clause } T \longrightarrow \text{trail } y \models_{as} C\text{Not } T$  and  
*z*: *trail* *z* = *c'* @ *Marked* *Kh* *i* # *H*  
**shows** *D* + {#*L*#}  $\notin$  *learned-clss* *z*  
**using** *assms*(1-2) *trM* *DL* *DH* *LH* *learned* *z*  
**proof** (*induction rule*: *cdcl<sub>W</sub>-o-induct-lev2*)  
**case** (*backtrack* *K* *j* *M1* *M2* *L'* *D'* *T*) **note** *decomp* = *this*(1) **and** *confl* = *this*(3) **and** *levD* = *this*(5)  
**and** *undef* = *this*(6) **and** *T* = *this*(7)  
**obtain** *M3* **where** *M3*: *trail* *y* = *M3* @ *M2* @ *Marked* *K* (*Suc* *j*) # *M1*  
**using** *decomp* *get-all-marked-decomposition-exists-prepend* **by** *metis*  
**have** *M*: *trail* *y* = *c* @ *Marked* *Kh* *i* # *H* **using** *trM* **by** *simp*  
**have** *H*: *get-all-levels-of-marked* (*trail* *y*) = *rev* [1..*l* + *backtrack-lvl* *y*]  
**using** *lev* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**have** *c'* @ *Marked* *Kh* *i* # *H* = *Propagated* *L'* (*D'* + {#*L'*#}) # *trail* (*reduce-trail-to* *M1* *y*)  
**using** *backtrack.premis*(6) *decomp* *undef* *T* *lev* **by** (*force* *simp*: *cdcl<sub>W</sub>-M-level-inv-def*)  
**then obtain** *d* **where** *d*: *M1* = *d* @ *Marked* *Kh* *i* # *H*  
**by** (*metis* (*no-types*) *decomp* *in-get-all-marked-decomposition-trail-update-trail* *list.inject*  
*list.sel*(3) *marked-lit.distinct*(1) *self-append-conv2* *tl-append2*)  
**have** *i*  $\in$  *set* (*get-all-levels-of-marked* (*M3* @ *M2* @ *Marked* *K* (*Suc* *j*) # *d* @ *Marked* *Kh* *i* # *H*))  
**by** *auto*  
**then have** *i* > 0 **unfolding** *H*[*unfolded* *M3* *d*] **by** *auto*  
**show** ?*case*  
**proof**  
**assume** *D* + {#*L*#}  $\in$  *learned-clss* *T*  
**then have** *DLD'*: *D* + {#*L*#} = *D'* + {#*L'*#}  
**using** *DL* *T* *neq0-conv* *undef* *decomp* *lev* **by** (*fastforce* *simp*: *cdcl<sub>W</sub>-M-level-inv-def*)  
**have** *L-cKh*: *atm-of* *L*  $\in$  *atm-of* 'lits-of (*c* @ [*Marked* *Kh* *i*])  
**using** *LH* *learned* *M* *DLD'*[*symmetric*] *confl* **by** (*fastforce* *simp* *add*: *image-iff*)  
**have** *get-all-levels-of-marked* (*M3* @ *M2* @ *Marked* *K* (*j* + 1) # *M1*)  
= *rev* [1..*l* + *backtrack-lvl* *y*]  
**using** *lev* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* *M3* **by** *auto*  
**from** *arg-cong*[*OF* *this*, *of*  $\lambda a. (Suc\ j) \in set\ a$ ] **have** *backtrack-lvl* *y*  $\geq$  *j* **by** *auto*  
  
**have** *DD'*[*simp*]: *D* = *D'*  
**proof** (*rule* *ccontr*)  
**assume** *D*  $\neq$  *D'*  
**then have** *L' ∈ # D* **using** *DLD'* **by** (*metis* *add.left-neutral* *count-single* *count-union*  
*diff-union-cancelR* *neq0-conv* *union-single-eq-member*)  
**then have** *get-level* *L'* (*trail* *y*)  $\leq$  *get-maximum-level* *D* (*trail* *y*)  
**using** *get-maximum-level-ge-get-level* **by** *blast*  
**moreover** {  
**have** *get-maximum-level* *D* (*trail* *y*) = *get-maximum-level* *D* *H*  
**using** *DH* **unfolding** *M* **by** (*simp* *add*: *get-maximum-level-skip-beginning*)  
**moreover**  
**have** *get-all-levels-of-marked* (*trail* *y*) = *rev* [1..*l* + *backtrack-lvl* *y*]  
**using** *lev* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**then have** *get-all-levels-of-marked* *H* = *rev* [1..*i*]  
**unfolding** *M* **by** (*auto* *dest*: *append-cons-eq-upt-length-i*  
*simp* *add*: *rev-swap*[*symmetric*])  
**then have** *get-maximum-possible-level* *H* < *i*  
**using** *get-maximum-possible-level-max-get-all-levels-of-marked*[*of* *H*] (*i* > 0) **by** *auto*  
**ultimately have** *get-maximum-level* *D* (*trail* *y*) < *i*  
**by** (*metis* (*full-types*) *dual-order.strict-trans* *nat-neq-iff* *not-le*  
*get-maximum-possible-level-ge-get-maximum-level*) }  
**moreover**

```

have L ∈# D'
  by (metis DLD' (D ≠ D') add.left-neutral count-single count-union diff-union-cancelR
    neq0-conv union-single-eq-member)
then have get-maximum-level D' (trail y) ≥ get-level L (trail y)
  using get-maximum-level-ge-get-level by blast
moreover {
  have get-all-levels-of-marked (c @ [Marked Kh i]) = rev [i..< backtrack-lvl y+1]
    using append-cons-eq-upt-length-i-end[of rev (get-all-levels-of-marked H) i
      rev (get-all-levels-of-marked c) Suc 0 Suc (backtrack-lvl y)] H
  unfolding M apply (auto simp add: rev-swap[symmetric])
    by (metis (no-types, hide-lams) Nil-is-append-conv Suc-le-eq less-Suc-eq list.sel(1)
      rev.simps(2) rev-rev-ident upt-Suc upt-rec)
  have get-level L (trail y) = get-level L (c @ [Marked Kh i])
    using L-cKh LH unfolding M by simp
  have get-level L (c @ [Marked Kh i]) ≥ i
    using L-cKh
      (get-all-levels-of-marked (c @ [Marked Kh i]) = rev [i..< backtrack-lvl y + 1])
      backtrack.hyps(2) calculation(1,2) by auto
  then have get-level L (trail y) ≥ i
    using M (get-level L (trail y) = get-level L (c @ [Marked Kh i])) by auto }
moreover have get-maximum-level D' (trail y) < get-level L' (trail y)
  using (j ≤ backtrack-lvl y) backtrack.hyps(2,5) calculation(1-4) by linarith
ultimately show False using backtrack.hyps(4) by linarith
qed
then have LL': L = L' using DLD' by auto
have nd: no-dup (trail y) using lev unfolding cdclW-M-level-inv-def by auto

{ assume D: D' = {#}
  then have j: j = 0 using levD by auto
  have ∀ m ∈ set M1. ¬is-marked m
    using H unfolding M3 j
    by (auto simp add: rev-swap[symmetric] get-all-levels-of-marked-no-marked
      dest!: append-cons-eq-upt-length-i)
  then have False using d by auto
}
moreover {
  assume D[simp]: D' ≠ {#}
  have i ≤ j
    using H unfolding M3 d by (auto simp add: rev-swap[symmetric]
      dest: upt-decomp-lt)
  have j > 0 apply (rule ccontr)
    using H (i > 0) unfolding M3 d
    by (auto simp add: rev-swap[symmetric] dest!: upt-decomp-lt)
  obtain L'' where
    L'' ∈# D' and
    L''D': get-level L'' (trail y) = get-maximum-level D' (trail y)
    using get-maximum-level-exists-lit-of-max-level[OF D, of trail y] by auto
  have L''M: atm-of L'' ∈ atm-of ' lits-of (trail y)
    using get-rev-level-ge-0-atm-of-in[of 0 L'' rev (trail y)] (j > 0) levD L''D' by auto
  then have L'' ∈ lits-of (Marked Kh i # d)
  proof -
    {
      assume L''H: atm-of L'' ∈ atm-of ' lits-of H
      have get-all-levels-of-marked H = rev [1..<i]
        using H unfolding M

```

```

    by (auto simp add: rev-swap[symmetric] dest!: append-cons-eq-upt-length-i)
  moreover have get-level L'' (trail y) = get-level L'' H
    using L''H unfolding M by simp
  ultimately have False
    using levD ⟨j>0⟩ get-rev-level-in-levels-of-marked[of L'' 0 rev H] ⟨i ≤ j⟩
    unfolding L''D'[symmetric] nd by auto
}
then show ?thesis
  using DD' DH ⟨L'' ∈ # D'⟩ atm-of-lit-in-atms-of contra-subsetD by metis
qed
then have False
  using DH ⟨L'' ∈ # D'⟩ nd unfolding M3 d
  by (auto simp add: atms-of-def image-iff image-subset-iff lits-of-def)
}
ultimately show False by blast
qed
qed auto

```

**lemma** *cdcl<sub>W</sub>-stgy-with-trail-end-has-not-been-learned*:

```

  assumes cdclW-stgy y z and
    cdclW-M-level-inv y and
    trail y = c @ Marked Kh i # H and
    D + {#L#} ∉ # learned-clss y and
    DH: atms-of D ⊆ atm-of 'lits-of H and
    LH: atm-of L ∉ atm-of 'lits-of H and
    ∀ T. conflicting y = C-Clause T ⟶ trail y ⊨as CNot T and
    trail z = c' @ Marked Kh i # H
  shows D + {#L#} ∉ # learned-clss z
  using assms
proof induction
  case conflict'
  then show ?case
    unfolding full1-def using tranclp-cdclW-cp-learned-clause-inv by auto
next
  case (other' T U) note o = this(1) and cp = this(3) and lev = this(4) and trY = this(5) and
    notin = this(6) and DH = this(7) and LH = this(8) and confl = this(9) and trU = this(10)
  obtain c' where c': trail T = c' @ Marked Kh i # H
    using cp beginning-not-marked-invert[of - trail T c' Kh i H]
    rtranclp-cdclW-cp-dropWhile-trail[of T U] unfolding trU full-def by fastforce
  show ?case
    using cdclW-o-cannot-learn[OF o lev trY notin DH LH confl c']
    rtranclp-cdclW-cp-learned-clause-inv cp unfolding full-def by auto
qed

```

**lemma** *rtranclp-cdcl<sub>W</sub>-stgy-with-trail-end-has-not-been-learned*:

```

  assumes (λa b. cdclW-stgy a b ∧ (∃ c. trail a = c @ Marked K i # H @ []))** S z and
    cdclW-all-struct-inv S and
    trail S = c @ Marked K i # H and
    D + {#L#} ∉ # learned-clss S and
    DH: atms-of D ⊆ atm-of 'lits-of H and
    LH: atm-of L ∉ atm-of 'lits-of H and
    ∃ c'. trail z = c' @ Marked K i # H
  shows D + {#L#} ∉ # learned-clss z
  using assms(1-4,7)
proof (induction rule: rtranclp-induct)

```

```

case base
then show ?case by auto[1]
next
case (step T U) note st = this(1) and s = this(2) and IH = this(3)[OF this(4-6)]
  and lev = this(4) and trS = this(5) and DL-S = this(6) and trU = this(7)
obtain c where c: trail T = c @ Marked K i # H using s by auto
obtain c' where c': trail U = c' @ Marked K i # H using trU by blast
have cdclW** S T
proof -
  have  $\forall p \text{ } pa. \exists s \text{ } sa. \forall sb \text{ } sc \text{ } sd \text{ } se. (\neg p^{**} (sb::'st) \text{ } sc \vee p \text{ } s \text{ } sa \vee pa^{**} \text{ } sb \text{ } sc)$ 
     $\wedge (\neg pa \text{ } s \text{ } sa \vee \neg p^{**} \text{ } sd \text{ } se \vee pa^{**} \text{ } sd \text{ } se)$ 
    by (metis (no-types) mono-rtrancp)
  then have cdclW-stgy** S T
    using st by blast
  then show ?thesis
    using rtrancp-cdclW-stgy-rtrancp-cdclW by blast
qed
then have lev': cdclW-all-struct-inv T
  using rtrancp-cdclW-all-struct-inv-inv[of S T] lev by auto
then have confl':  $\forall Ta. \text{conflicting } T = C\text{-Clause } Ta \longrightarrow \text{trail } T \models_{as} C\text{Not } Ta$ 
  unfolding cdclW-all-struct-inv-def cdclW-conflicting-def by blast
show ?case
  apply (rule cdclW-stgy-with-trail-end-has-not-been-learned[OF - - c - DH LH confl' c'])
  using s lev' IH c unfolding cdclW-all-struct-inv-def by blast+
qed

lemma cdclW-stgy-new-learned-clause:
assumes cdclW-stgy S T and
  lev: cdclW-M-level-inv S and
  E  $\notin \#$  learned-clss S and
  E  $\in \#$  learned-clss T
shows  $\exists S'. \text{backtrack } S \text{ } S' \wedge \text{conflicting } S = C\text{-Clause } E \wedge \text{full } cdcl_W\text{-cp } S' \text{ } T$ 
using assms
proof induction
case conflict'
then show ?case unfolding full1-def by (auto dest: trancp-cdclW-cp-learned-clause-inv)
next
case (other' T U) note o = this(1) and cp = this(3) and not-yet = this(5) and learned = this(6)
have E  $\in \#$  learned-clss T
  using learned cp rtrancp-cdclW-cp-learned-clause-inv unfolding full-def by auto
then have backtrack S T and conflicting S = C-Clause E
  using cdclW-o-new-clause-learned-is-backtrack-step[OF - not-yet o] lev by blast+
then show ?case using cp by blast
qed

lemma cdclW-stgy-no-relearned-clause:
assumes
  invR: cdclW-all-struct-inv R and
  st': cdclW-stgy** R S and
  bt: backtrack S T and
  confl: conflicting S = C-Clause E and
  already-learned: E  $\in \#$  clauses S and
  R: trail R = []
shows False
proof -

```



**have**  $M\text{-lev}$ :  $cdcl_W\text{-}M\text{-level-inv } R$   
**using**  $invR$  **unfolding**  $cdcl_W\text{-all-struct-inv-def}$  **by**  $auto$   
**have**  $cdcl_W\text{-}M\text{-level-inv } S$   
**using**  $M\text{-lev}$   $assms(2)$   $rtrancp\text{-}cdcl_W\text{-stgy-consistent-inv}$  **by**  $blast$   
**with**  $bt$  **obtain**  $D L M1 M2\text{-loc } K i$  **where**  
 $T$ :  $T \sim cons\text{-trail } (Propagated L ((D + \{\#L\#\})))$   
 $(reduce\text{-trail-to } M1 (add\text{-learned-cl } (D + \{\#L\#\}))$   
 $(update\text{-backtrack-lvl } (get\text{-maximum-level } D (trail S)) (update\text{-conflicting } C\text{-True } S)))$   
**and**  
 $decomp$ :  $(Marked K (Suc (get\text{-maximum-level } D (trail S))) \# M1, M2\text{-loc}) \in$   
 $set (get\text{-all-marked-decomposition } (trail S))$  **and**  
 $k$ :  $get\text{-level } L (trail S) = backtrack\text{-lvl } S$  **and**  
 $level$ :  $get\text{-level } L (trail S) = get\text{-maximum-level } (D + \{\#L\#\}) (trail S)$  **and**  
 $confl\text{-}S$ :  $conflicting S = C\text{-Clause } (D + \{\#L\#\})$  **and**  
 $i$ :  $i = get\text{-maximum-level } D (trail S)$  **and**  
 $undef$ :  $undefined\text{-lit } M1 L$   
**by**  $(induction\ rule: backtrack\text{-induction-lev2})$   $metis$   
**obtain**  $M2$  **where**  
 $M$ :  $trail S = M2 @ Marked K (Suc i) \# M1$   
**using**  $get\text{-all-marked-decomposition-exists-prepend}[OF decomp]$  **unfolding**  $i$  **by**  $(metis\ append\text{-assoc})$

**have**  $invS$ :  $cdcl_W\text{-all-struct-inv } S$   
**using**  $invR$   $rtrancp\text{-}cdcl_W\text{-all-struct-inv-inv } rtrancp\text{-}cdcl_W\text{-stgy-rtrancp-cdcl}_W$   $st'$  **by**  $blast$   
**then have**  $conf$ :  $cdcl_W\text{-conflicting } S$  **unfolding**  $cdcl_W\text{-all-struct-inv-def}$  **by**  $blast$   
**then have**  $trail S \models_{as} CNot (D + \{\#L\#\})$  **unfolding**  $cdcl_W\text{-conflicting-def } confl\text{-}S$  **by**  $auto$   
**then have**  $MD$ :  $trail S \models_{as} CNot D$  **by**  $auto$

**have**  $lev'$ :  $cdcl_W\text{-}M\text{-level-inv } S$  **using**  $invS$  **unfolding**  $cdcl_W\text{-all-struct-inv-def}$  **by**  $blast$

**have**  $get\text{-lvl}\text{-}M$ :  $get\text{-all-levels-of-marked } (trail S) = rev [1..<Suc (backtrack\text{-lvl } S)]$   
**using**  $lev'$  **unfolding**  $cdcl_W\text{-}M\text{-level-inv-def}$  **by**  $auto$

**have**  $lev$ :  $cdcl_W\text{-}M\text{-level-inv } R$  **using**  $invR$  **unfolding**  $cdcl_W\text{-all-struct-inv-def}$  **by**  $blast$   
**then have**  $vars\text{-of-}D$ :  $atms\text{-of } D \subseteq atm\text{-of } 'lits\text{-of } M1$   
**using**  $backtrack\text{-atms-of-}D\text{-in-}M1[OF lev' undef - decomp - - - T]$   $confl\text{-}S$   $conf T$   $decomp k level$   
 $lev' i undef$  **unfolding**  $cdcl_W\text{-conflicting-def}$  **by**  $(auto\ simp: cdcl_W\text{-}M\text{-level-inv-def})$   
**have**  $no\text{-dup}$   $(trail S)$  **using**  $lev'$  **by**  $(auto\ simp: cdcl_W\text{-}M\text{-level-inv-decomp})$   
**have**  $vars\text{-in-}M1$ :  
 $\forall x \in atms\text{-of } D. x \notin atm\text{-of } 'lits\text{-of } (M2 @ [Marked K (get\text{-maximum-level } D (trail S) + 1)])$   
**apply**  $(rule\ vars\text{-of-}D\ distinct\text{-atms-of-incl-not-in-other}[of$   
 $M2 @ Marked K (get\text{-maximum-level } D (trail S) + 1) \# [] M1 D])$   
**using**  $\langle no\text{-dup } (trail S) \rangle M vars\text{-of-}D$  **by**  $simp\text{-all}$   
**have**  $M1\text{-}D$ :  $M1 \models_{as} CNot D$   
**using**  $vars\text{-in-}M1$   $true\text{-annots-remove-if-notin-vars}[of M2 @ Marked K (i + 1) \# [] M1 CNot D]$   
 $\langle trail S \models_{as} CNot D \rangle M$  **by**  $simp$

**have**  $get\text{-lvl}\text{-}M$ :  $get\text{-all-levels-of-marked } (trail S) = rev [1..<Suc (backtrack\text{-lvl } S)]$   
**using**  $lev'$  **unfolding**  $cdcl_W\text{-}M\text{-level-inv-def}$  **by**  $auto$   
**then have**  $backtrack\text{-lvl } S > 0$  **unfolding**  $M$  **by**  $(auto\ split: split\text{-if-asm } simp\ add: upt.simps(2))$

**obtain**  $M1' K' Ls$  **where**  
 $M'$ :  $trail S = Ls @ Marked K' (backtrack\text{-lvl } S) \# M1'$  **and**  
 $Ls$ :  $\forall l \in set Ls. \neg is\text{-marked } l$  **and**  
 $set M1 \subseteq set M1'$   
**proof** –

```

let ?Ls = takeWhile (Not o is-marked) (trail S)
have MLs: trail S = ?Ls @ dropWhile (Not o is-marked) (trail S)
  by auto
have dropWhile (Not o is-marked) (trail S) ≠ [] unfolding M by auto
moreover
  from hd-dropWhile[OF this] have is-marked(hd (dropWhile (Not o is-marked) (trail S)))
    by simp
ultimately
  obtain K' K'k where
    K'k: dropWhile (Not o is-marked) (trail S)
      = Marked K' K'k # tl (dropWhile (Not o is-marked) (trail S))
  by (cases dropWhile (Not o is-marked) (trail S);
      cases hd (dropWhile (Not o is-marked) (trail S)))
    simp-all
moreover have ∀ l ∈ set ?Ls. ¬is-marked l using set-takeWhileD by force
moreover
  have get-all-levels-of-marked (trail S)
    = K'k # get-all-levels-of-marked(tl (dropWhile (Not o is-marked) (trail S)))
  apply (subst MLs, subst K'k)
  using calculation(2) by (auto simp add: get-all-levels-of-marked-no-marked)
  then have K'k = backtrack-lvl S
  using calculation(2) by (auto split: split-if-asm simp add: get-lvls-M upt.simps(2))
moreover have set M1 ⊆ set (tl (dropWhile (Not o is-marked) (trail S)))
  unfolding M by (induction M2) auto
ultimately show ?thesis using that MLs by metis
qed

have get-lvls-M: get-all-levels-of-marked (trail S) = rev [1..W-M-level-inv-def by auto
then have backtrack-lvl S > 0 unfolding M by (auto split: split-if-asm simp add: upt.simps(2) i)

have M1'-D: M1' ⊨as CNot D using M1-D ⟨set M1 ⊆ set M1'⟩ by (auto intro: true-annots-mono)
have -L ∈ lits-of (trail S) using conf confl-S unfolding cdclW-conflicting-def by auto
have lvls-M1': get-all-levels-of-marked M1' = rev [1..W-stgy** R Y and
  YZ: cdclW-stgy Y Z and
  nt: ¬ (∃ c. trail Y = c @ Marked K' (backtrack-lvl S) # M1' @ []) and
  Z: (λa b. cdclW-stgy a b ∧ (∃ c. trail a = c @ Marked K' (backtrack-lvl S) # M1' @ []))**
    Z S
using rtranclp-cdclW-new-marked-at-beginning-is-decide'[OF st' - lev, of Ls K']
  backtrack-lvl S M1' []
unfolding R M' by auto
have [simp]: cdclW-M-level-inv Y

```

```

    using RY lev rtrancpl-cdclW-stgy-consistent-inv by blast
  obtain M' where trZ: trail Z = M' @ Marked K' (backtrack-lvl S) # M1'
    using rtrancpl-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
  have no-dup (trail Y)
    using RY lev rtrancpl-cdclW-stgy-consistent-inv unfolding cdclW-M-level-inv-def by blast
  then obtain Y' where
    dec: decide Y Y' and
    Y'Z: full cdclW-cp Y' Z and
    no-step cdclW-cp Y
    using cdclW-stgy-trail-has-new-marked-is-decide-step[OF YZ nt Z] M' by auto
  have trY: trail Y = M1'
  proof -
    obtain M' where M: trail Z = M' @ Marked K' (backtrack-lvl S) # M1'
      using rtrancpl-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
    obtain M'' where M'': trail Z = M'' @ trail Y' and  $\forall m \in \text{set } M''. \neg \text{is-marked } m$ 
      using Y'Z rtrancpl-cdclW-cp-dropWhile-trail' unfolding full-def by blast
    obtain M''' where trail Y' = M''' @ Marked K' (backtrack-lvl S) # M1'
      using M'' unfolding M
      by (metis (no-types, lifting)  $\forall m \in \text{set } M''. \neg \text{is-marked } m$  beginning-not-marked-invert)
    then show ?thesis using dec nt by (induction M''') auto
  qed
  have Y-CT: conflicting Y = C-True using  $\langle \text{decide } Y Y' \rangle$  by auto
  have cdclW** R Y by (simp add: RY rtrancpl-cdclW-stgy-rtrancpl-cdclW)
  then have init-clss Y = init-clss R using rtrancpl-cdclW-init-clss[of R Y] M-lev by auto
  { assume DL: D + {#L#}  $\in$  # clauses Y
    have atm-of L  $\notin$  atm-of ' lits-of M1
      apply (rule backtrack-lit-skipped[of - S])
      using decomp i k lev' unfolding cdclW-M-level-inv-def by auto
    then have LM1: undefined-lit M1 L
      by (metis Marked-Propagated-in-iff-in-lits-of atm-of-uminus image-eqI)
    have L-trY: undefined-lit (trail Y) L
      using L-notin  $\langle \text{no-dup } (\text{trail } S) \rangle$  unfolding defined-lit-map trY M'
      by (auto simp add: image-iff lits-of-def)
    have  $\exists Y'. \text{propagate } Y Y'$ 
      using propagate-rule[of Y] DL M1'-D L-trY Y-CT trY DL by (metis state-eq-ref)
    then have False using  $\langle \text{no-step cdcl}_W\text{-cp } Y \rangle$  propagate' by blast
  }
  moreover {
    assume DL: D + {#L#}  $\notin$  # clauses Y
    have lY-lZ: learned-clss Y = learned-clss Z
      using dec Y'Z rtrancpl-cdclW-cp-learned-clause-inv[of Y' Z] unfolding full-def
      by auto
    have invZ: cdclW-all-struct-inv Z
      by (meson RY YZ invR r-into-rtrancpl rtrancpl-cdclW-all-struct-inv-inv
          rtrancpl-cdclW-stgy-rtrancpl-cdclW)
    have D + {#L#}  $\notin$  #learned-clss S
      apply (rule rtrancpl-cdclW-stgy-with-trail-end-has-not-been-learned[OF Z invZ trZ])
      using DL lY-lZ unfolding clauses-def apply simp
      apply (metis (no-types, lifting)  $\langle \text{set } M1 \subseteq \text{set } M1' \rangle$  image-mono order-trans
          vars-of-D lits-of-def)
      using L-notin  $\langle \text{no-dup } (\text{trail } S) \rangle$  unfolding M' by (auto simp add: image-iff lits-of-def)
    then have False
      using already-learned DL confl st' M-lev unfolding M'
      by (simp add:  $\langle \text{init-clss } Y = \text{init-clss } R \rangle$  clauses-def confl-S
          rtrancpl-cdclW-stgy-no-more-init-clss)
  }

```

```

}
ultimately show False by blast
qed

lemma rtrancp-cdclW-stgy-distinct-mset-clauses:
  assumes
    invR: cdclW-all-struct-inv R and
    st: cdclW-stgy** R S and
    dist: distinct-mset (clauses R) and
    R: trail R = []
  shows distinct-mset (clauses S)
  using st
proof (induction)
  case base
  then show ?case using dist by simp
next
  case (step S T) note st = this(1) and s = this(2) and IH = this(3)
  from s show ?case
  proof (cases rule: cdclW-stgy.cases)
    case conflict'
    then show ?thesis
      using IH unfolding full1-def by (auto dest: trancp-cdclW-cp-no-more-clauses)
  next
    case (other' S') note o = this(1) and full = this(3)
    have [simp]: clauses T = clauses S'
      using full unfolding full-def by (auto dest: rtrancp-cdclW-cp-no-more-clauses)
    show ?thesis
      using o IH
    proof (cases rule: cdclW-o-rule-cases)
      case backtrack
      moreover
        have cdclW-all-struct-inv S
          using invR rtrancp-cdclW-stgy-cdclW-all-struct-inv st by blast
        then have cdclW-M-level-inv S
          unfolding cdclW-all-struct-inv-def by auto
        ultimately obtain E where
          conflicting S = C-Clause E and
          cls-S': clauses S' = {#E#} + clauses S
          using cdclW-M-level-inv S
          by (induction rule: backtrack-induction-lev2) (auto simp: cdclW-M-level-inv-decomp)
        then have E ∉ clauses S
          using cdclW-stgy-no-relearned-clause R invR local.backtrack st by blast
        then show ?thesis using IH by (simp add: distinct-mset-add-single cls-S')
      qed auto
    qed
  qed

```

```

lemma cdclW-stgy-distinct-mset-clauses:
  assumes
    st: cdclW-stgy** (init-state N) S and
    no-duplicate-clause: distinct-mset N and
    no-duplicate-in-clause: distinct-mset-mset N
  shows distinct-mset (clauses S)
  using rtrancp-cdclW-stgy-distinct-mset-clauses[OF - st] assms
  by (auto simp: cdclW-all-struct-inv-def distinct-cdclW-state-def)

```

## 17.9 Decrease of a measure

**fun** *cdcl<sub>W</sub>-measure* **where**

```
cdclW-measure S =
  [(3::nat) ^ (card (atms-of-mu (init-clss S))) - card (set-mset (learned-clss S)),
   if conflicting S = C-True then 1 else 0,
   if conflicting S = C-True then card (atms-of-mu (init-clss S)) - length (trail S)
   else length (trail S)
  ]
```

**lemma** *length-model-le-vars-all-inv*:

```
assumes cdclW-all-struct-inv S
shows length (trail S) ≤ card (atms-of-mu (init-clss S))
using assms length-model-le-vars[of S] unfolding cdclW-all-struct-inv-def
by (auto simp: cdclW-M-level-inv-decomp)
end
```

**locale** *cdcl<sub>W</sub>-termination* =

```
cdclW-ops trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
add-init-cls
add-learned-cls remove-cls update-backtrack-lvl update-conflicting init-state
restart-state
for
  trail :: 'st::equal ⇒ ('v::linorder, nat, 'v clause) marked-lits and
  init-clss :: 'st ⇒ 'v clauses and
  learned-clss :: 'st ⇒ 'v clauses and
  backtrack-lvl :: 'st ⇒ nat and
  conflicting :: 'st ⇒ 'v clause conflicting-clause and

  cons-trail :: ('v, nat, 'v clause) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-init-cls :: 'v clause ⇒ 'st ⇒ 'st and
  add-learned-cls :: 'v clause ⇒ 'st ⇒ 'st and
  remove-cls :: 'v clause ⇒ 'st ⇒ 'st and
  update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
  update-conflicting :: 'v clause conflicting-clause ⇒ 'st ⇒ 'st and

  init-state :: 'v clauses ⇒ 'st and
  restart-state :: 'st ⇒ 'st
```

**begin**

**lemma** *learned-clss-less-upper-bound*:

```
fixes S :: 'st
assumes
  distinct-cdclW-state S and
  ∀ s ∈ # learned-clss S. ¬tautology s
shows card(set-mset (learned-clss S)) ≤ 3 ^ card (atms-of-mu (learned-clss S))
proof -
have set-mset (learned-clss S) ⊆ build-all-simple-clss (atms-of-mu (learned-clss S))
apply (rule simplified-in-build-all)
using assms unfolding distinct-cdclW-state-def by auto
then have card(set-mset (learned-clss S))
  ≤ card (build-all-simple-clss (atms-of-mu (learned-clss S)))
by (simp add: build-all-simple-clss-finite card-mono)
then show ?thesis
by (meson atms-of-m-finite build-all-simple-clss-card finite-set-mset order-trans)
```

qed

**lemma** *lexn3[intro!, simp]:*

$a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$   
 $\implies ([a::nat, b, c], [a', b', c']) \in lexn \{(x, y). x < y\} \text{ ?}$

**apply** *auto*

**unfolding** *lexn-conv* **apply** *fastforce*

**unfolding** *lexn-conv* **apply** *auto*

**apply** (*metis append.simps(1) append.simps(2)*) +

**done**

**lemma** *cdcl<sub>W</sub>-measure-decreasing:*

**fixes** *S :: 'st*

**assumes**

*cdcl<sub>W</sub> S S' and*

*no-restart:*

$\neg(\text{learned-clss } S \subseteq \# \text{ learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = C\text{-True})$

**and**

*learned-clss S ⊆# learned-clss S' and*

*no-relearn:  $\bigwedge S'. \text{backtrack } S S' \implies \forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow T \notin \# \text{ learned-clss } S$*

**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** (*cdcl<sub>W</sub>-measure S', cdcl<sub>W</sub>-measure S*)  $\in lexn \{(a, b). a < b\} \text{ ?}$

**using** *assms(1) M-level assms(2,3)*

**proof** (*induct rule: cdcl<sub>W</sub>-all-induct-lev2*)

**case** (*propagate C L*) **note** *undef = this(3) and T = this(4) and conf = this(5)*

**have** *propa: propagate S (cons-trail (Propagated L (C + {#L#}))) S*

**using** *propagate-rule[OF - propagate.hyps(1,2)] propagate.hyps by auto*

**then have** *no-dup': no-dup (Propagated L (C + {#L#})) # trail S*

**by** (*metis M-level cdcl<sub>W</sub>-M-level-inv-decomp(2) marked-lit.sel(2) propagate'*

*r-into-rtranclp rtranclp-cdcl<sub>W</sub>-cp-consistent-inv trail-cons-trail undef*)

**let** *?N = init-clss S*

**have** *no-strange-atm (cons-trail (Propagated L (C + {#L#}))) S*

**using** *alien cdcl<sub>W</sub>.propagate cdcl<sub>W</sub>-no-strange-atm-inv propa M-level by blast*

**then have** *atm-of ' lits-of (Propagated L (C + {#L#})) # trail S*

$\subseteq \text{atms-of-mu } (\text{init-clss } S)$

**using** *undef unfolding no-strange-atm-def by auto*

**then have** *card (atm-of ' lits-of (Propagated L (C + {#L#})) # trail S)*

$\leq \text{card } (\text{atms-of-mu } (\text{init-clss } S))$

**by** (*meson atms-of-m-finite card-mono finite-set-mset*)

**then have** *length (Propagated L (C + {#L#})) # trail S ≤ card (atms-of-mu ?N)*

**using** *no-dup-length-eq-card-atm-of-lits-of no-dup' by fastforce*

**then have** *H: card (atms-of-mu (init-clss S)) - length (trail S)*

$= \text{Suc } (\text{card } (\text{atms-of-mu } (\text{init-clss } S)) - \text{Suc } (\text{length } (\text{trail } S)))$

**by** *simp*

**show** *?case using conf T undef by (auto simp: H)*

**next**

**case** (*decide L*) **note** *conf = this(1) and undef = this(2) and T = this(4)*

**moreover**

**have** *dec: decide S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))*

```

    using decide.intros decide.hyps by force
  then have cdclW:cdclW S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))
    using cdclW.simps by blast
moreover
  have lev: cdclW-M-level-inv (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))
    using cdclW M-level cdclW-consistent-inv[OF cdclW] by auto
  then have no-dup: no-dup (Marked L (backtrack-lvl S + 1) # trail S)
    using undef unfolding cdclW-M-level-inv-def by auto
  have no-strange-atm (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))
    using M-level alien calculation(4) cdclW-no-strange-atm-inv by blast
  then have length (Marked L ((backtrack-lvl S) + 1) # (trail S))
    ≤ card (atms-of-mu (init-clss S))
    using no-dup clauses-def undef
    length-model-le-vars[of cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S)]
    by fastforce
  ultimately show ?case using conf by auto
next
  case (skip L C' M D) note tr = this(1) and conf = this(2) and T = this(5)
  show ?case using conf T unfolding clauses-def by (simp add: tr)
next
  case conflict
  then show ?case by simp
next
  case resolve
  then show ?case using finite unfolding clauses-def by simp
next
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and conf = this(3) and undef = this(6)
  and
    T = this(7) and lev = this(8)
  let ?S' = T
  have bt: backtrack S ?S'
    using backtrack.hyps backtrack.intros[of S - - - D L K i] by auto
  have D + {#L#} ∉ learned-clss S
    using no-relearn conf bt by auto
  then have card-T:
    card (set-mset ({#D + {#L#}#} + learned-clss S)) = Suc (card (set-mset (learned-clss S)))
    by (simp add:)
  have distinct-cdclW-state ?S'
    using bt M-level distinct-cdclW-state-inv no-dup other by blast
  moreover have ∀ s ∈ #learned-clss ?S'. ¬ tautology s
    using learned-clss-are-not-tautologies[OF cdclW.other[OF cdclW-o.bj[OF
      cdclW-bj.backtrack[OF bt]]]] M-level no-taut confl by auto
  ultimately have card (set-mset (learned-clss T)) ≤ 3 ^ card (atms-of-mu (learned-clss T))
    by (auto simp: clauses-def learned-clss-less-upper-bound)
  then have H: card (set-mset ({#D + {#L#}#} + learned-clss S))
    ≤ 3 ^ card (atms-of-mu ({#D + {#L#}#} + learned-clss S))
    using T undef decomp lev by (auto simp: cdclW-M-level-inv-decomp)
  moreover
    have atms-of-mu ({#D + {#L#}#} + learned-clss S) ⊆ atms-of-mu (init-clss S)
      using alien conf unfolding no-strange-atm-def by auto
    then have card-f: card (atms-of-mu ({#D + {#L#}#} + learned-clss S))
      ≤ card (atms-of-mu (init-clss S))
      by (meson atms-of-m-finite card-mono finite-set-mset)
    then have (3::nat) ^ card (atms-of-mu ({#D + {#L#}#} + learned-clss S))
      ≤ 3 ^ card (atms-of-mu (init-clss S)) by simp

```

```

ultimately have ( $\beta :: \text{nat}$ )  $\wedge$  card (atms-of-mu (init-clss S))
   $\geq$  card (set-mset ( $\{\#D + \{\#L\}\#\} + \text{learned-clss S}$ ))
  using le-trans by blast
then show ?case using decomp undef diff-less-mono2 card-T T lev
  by (auto simp: cdclW-M-level-inv-decomp)
next
case restart
then show ?case using alien by (auto simp: state-eq-def simp del: state-simp)
next
case (forget C T)
then have  $C \in \# \text{learned-clss S}$  and  $C \notin \# \text{learned-clss T}$ 
  by auto
then show ?case using forget(9) by (simp add: mset-leD)
qed

```

```

lemma propagate-measure-decreasing:
  fixes S :: 'st
  assumes propagate S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S)  $\in \text{lexn } \{(a, b). a < b\} \text{ } 3$ 
  apply (rule cdclW-measure-decreasing)
  using assms(1) propagate apply blast
    using assms(1) apply (auto simp add: propagate.simps)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def)
  done

```

```

lemma conflict-measure-decreasing:
  fixes S :: 'st
  assumes conflict S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S)  $\in \text{lexn } \{(a, b). a < b\} \text{ } 3$ 
  apply (rule cdclW-measure-decreasing)
  using assms(1) conflict apply blast
    using assms(1) apply (auto simp add: propagate.simps)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def)
  done

```

```

lemma decide-measure-decreasing:
  fixes S :: 'st
  assumes decide S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S)  $\in \text{lexn } \{(a, b). a < b\} \text{ } 3$ 
  apply (rule cdclW-measure-decreasing)
  using assms(1) decide other apply blast
    using assms(1) apply (auto simp add: propagate.simps)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def)
  done

```

```

lemma trans-le:
  trans  $\{(a, (b :: \text{nat})). a < b\}$ 
  unfolding trans-def by auto

```

```

lemma cdclW-cp-measure-decreasing:
  fixes S :: 'st
  assumes cdclW-cp S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S)  $\in \text{lexn } \{(a, b). a < b\} \text{ } 3$ 
  using assms
proof induction

```



```

  case conflict'
  then show ?case using conflict-measure-decreasing by blast
next
  case propagate'
  then show ?case using propagate-measure-decreasing by blast
qed

lemma tranclp-cdclW-cp-measure-decreasing:
  fixes S :: 'st
  assumes cdclW-cp++ S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S) ∈ lexn {(a, b). a < b} 3
  using assms
proof induction
  case base
  then show ?case using cdclW-cp-measure-decreasing by blast
next
  case (step T U) note st = this(1) and step = this(2) and IH = this(3) and inv = this(4)
  then have (cdclW-measure T, cdclW-measure S) ∈ lexn {a. case a of (a, b) ⇒ a < b} 3 by blast

  moreover have (cdclW-measure U, cdclW-measure T) ∈ lexn {a. case a of (a, b) ⇒ a < b} 3
  using cdclW-cp-measure-decreasing[OF step] rtranclp-cdclW-all-struct-inv-inv inv
  tranclp-cdclW-cp-tranclp-cdclW[OF st]
  unfolding trans-def rtranclp-unfold
  by blast
  ultimately show ?case using lexn-transI[OF trans-le] unfolding trans-def by blast
qed

lemma cdclW-stgy-step-decreasing:
  fixes R S T :: 'st
  assumes cdclW-stgy S T and
  cdclW-stgy** R S
  trail R = [] and
  cdclW-all-struct-inv R
  shows (cdclW-measure T, cdclW-measure S) ∈ lexn {(a, b). a < b} 3
proof -
  have cdclW-all-struct-inv S
  using assms
  by (metis rtranclp-unfold rtranclp-cdclW-all-struct-inv-inv tranclp-cdclW-stgy-tranclp-cdclW)
with assms show ?thesis
proof induction
  case (conflict' V) note cp = this(1) and inv = this(5)
  show ?case
    using tranclp-cdclW-cp-measure-decreasing[OF HOL.conjunct1[OF cp[unfolded full1-def]] inv]
    .
next
  case (other' T U) note st = this(1) and H = this(4,5,6,7) and cp = this(3)
  have cdclW-all-struct-inv T
  using cdclW-all-struct-inv-inv other other'.hyps(1) other'.prems(4) by blast
  from tranclp-cdclW-cp-measure-decreasing[OF - this]
  have le-or-eq: (cdclW-measure U, cdclW-measure T) ∈ lexn {a. case a of (a, b) ⇒ a < b} 3 ∨
  cdclW-measure U = cdclW-measure T
  using cp unfolding full-def rtranclp-unfold by blast
  moreover
  have cdclW-M-level-inv S
  using cdclW-all-struct-inv-def other'.prems(4) by blast

```

```

with st have (cdclW-measure T, cdclW-measure S)  $\in$  lern {a. case a of (a, b)  $\Rightarrow$  a < b}  $\exists$ 
proof (induction rule:cdclW-o-induct-lev2)
  case (decide T)
  then show ?case using decide-measure-decreasing H by blast
next
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and undef = this(6) and T =
this(7)
  have bt: backtrack S T
  apply (rule backtrack-rule)
  using backtrack.hyps by auto
  then have no-relearn:  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow T \notin \# \text{ learned-clss } S$ 
  using cdclW-stgy-no-relearned-clause[of R S T] H
  unfolding cdclW-all-struct-inv-def clauses-def by auto
  have inv: cdclW-all-struct-inv S
  using  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$  by blast
  show ?case
  apply (rule cdclW-measure-decreasing)
    using bt cdclW-bj.backtrack cdclW-o.bj other apply simp
    using bt T undef decomp inv unfolding cdclW-all-struct-inv-def
      cdclW-M-level-inv-def apply auto[]
    using bt T undef decomp inv unfolding cdclW-all-struct-inv-def
      cdclW-M-level-inv-def apply auto[]
    using bt no-relearn apply auto[]
    using inv unfolding cdclW-all-struct-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def by simp
  next
  case skip
  then show ?case by force
  next
  case resolve
  then show ?case by force
  qed
ultimately show ?case
  by (metis lern-transI transD trans-le)
qed
qed

```

```

lemma tranclp-cdclW-stgy-decreasing:
  fixes R S T :: 'st
  assumes cdclW-stgy++ R S
  trail R = [] and
  cdclW-all-struct-inv R
  shows (cdclW-measure S, cdclW-measure R)  $\in$  lern {(a, b). a < b}  $\exists$ 
  using assms
  apply induction
    using cdclW-stgy-step-decreasing[of R - R] apply blast
  using cdclW-stgy-step-decreasing[of - - R] tranclp-into-rtranclp[of cdclW-stgy R]
  lern-transI[OF trans-le, of  $\exists$ ] unfolding trans-def by blast

```

```

lemma tranclp-cdclW-stgy-S0-decreasing:
  fixes R S T :: 'st
  assumes pl: cdclW-stgy++ (init-state N) S and

```

```

no-dup: distinct-mset-mset N
shows (cdclW-measure S, cdclW-measure (init-state N)) ∈ leW {(a, b). a < b} 3
proof -
  have cdclW-all-struct-inv (init-state N)
    using no-dup unfolding cdclW-all-struct-inv-def by auto
  then show ?thesis using pl tranclp-cdclW-stgy-decreasing init-state-trail by blast
qed

lemma wf-tranclp-cdclW-stgy:
  wf {(S::'st, init-state N) | S N. distinct-mset-mset N ∧ cdclW-stgy++ (init-state N) S}
  apply (rule wf-wf-if-measure'-notation2[of leW {(a, b). a < b} 3 - - cdclW-measure])
  apply (simp add: wf wf-leW)
  using tranclp-cdclW-stgy-S0-decreasing by blast
end

end
theory DPLL-CDCL-W-Implementation
imports Partial-Annotated-Clausal-Logic
begin

```

## 18 Simple Implementation of the DPLL and CDCL

### 18.1 Common Rules

#### 18.1.1 Propagation

The following theorem holds:

```

lemma lits-of-unfold[iff]:
  (∀ c ∈ set C. - c ∈ lits-of Ms) ⟷ Ms ⊨as CNot (mset C)
  unfolding true-annot-def Ball-def true-annot-def CNot-def mem-set-multiset-eq by auto

```

The right-hand version is written at a high-level, but only the left-hand side is executable.

```

definition is-unit-clause :: 'a literal list ⇒ ('a, 'b, 'c) marked-lit list ⇒ 'a literal option
where
  is-unit-clause l M =
    (case List.filter (λa. atm-of a ∉ atm-of ' lits-of M) l of
     a # [] ⇒ if M ⊨as CNot (mset l - {#a#}) then Some a else None
     | - ⇒ None)

```

```

definition is-unit-clause-code :: 'a literal list ⇒ ('a, 'b, 'c) marked-lit list
  ⇒ 'a literal option where
  is-unit-clause-code l M =
    (case List.filter (λa. atm-of a ∉ atm-of ' lits-of M) l of
     a # [] ⇒ if (∀ c ∈ set (remove1 a l). - c ∈ lits-of M) then Some a else None
     | - ⇒ None)

```

```

lemma is-unit-clause-is-unit-clause-code[code]:
  is-unit-clause l M = is-unit-clause-code l M
proof -
  have 1: ⋀ a. (∀ c ∈ set (remove1 a l). - c ∈ lits-of M) ⟷ M ⊨as CNot (mset l - {#a#})
    using lits-of-unfold[of remove1 - l, of - M] by simp
  thus ?thesis
    unfolding is-unit-clause-code-def is-unit-clause-def 1 by blast
qed

```

**lemma** *is-unit-clause-some-undef*:

**assumes** *is-unit-clause*  $l\ M = \text{Some } a$

**shows** *undefined-lit*  $M\ a$

**proof** –

**have** (case  $[a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$  of  $[] \Rightarrow \text{None}$   
 $| [a] \Rightarrow \text{if } M \models_{\text{as}} \text{CNot } (\text{mset } l - \{\#a\# \}) \text{ then } \text{Some } a \text{ else } \text{None}$   
 $| a \# ab \# xa \Rightarrow \text{Map.empty } xa) = \text{Some } a$

**using** *assms* **unfolding** *is-unit-clause-def* .

**hence**  $a \in \text{set } [a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$

**apply** (case-tac  $[a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$ )

**apply** *simp*

**apply** (case-tac *list*) **by** (auto split: split-if-asm)

**hence**  $\text{atm-of } a \notin \text{atm-of ' lits-of } M$  **by** auto

**thus** ?thesis

**by** (simp add: Marked-Propagated-in-iff-in-lits-of  
 $\text{atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set}$  )

**qed**

**lemma** *is-unit-clause-some-CNot*:  $\text{is-unit-clause } l\ M = \text{Some } a \implies M \models_{\text{as}} \text{CNot } (\text{mset } l - \{\#a\# \})$

**unfolding** *is-unit-clause-def*

**proof** –

**assume** (case  $[a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$  of  $[] \Rightarrow \text{None}$   
 $| [a] \Rightarrow \text{if } M \models_{\text{as}} \text{CNot } (\text{mset } l - \{\#a\# \}) \text{ then } \text{Some } a \text{ else } \text{None}$   
 $| a \# ab \# xa \Rightarrow \text{Map.empty } xa) = \text{Some } a$

**thus** ?thesis

**apply** (case-tac  $[a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$ , *simp*)

**apply** *simp*

**apply** (case-tac *list*) **by** (auto split: split-if-asm)

**qed**

**lemma** *is-unit-clause-some-in*:  $\text{is-unit-clause } l\ M = \text{Some } a \implies a \in \text{set } l$

**unfolding** *is-unit-clause-def*

**proof** –

**assume** (case  $[a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$  of  $[] \Rightarrow \text{None}$   
 $| [a] \Rightarrow \text{if } M \models_{\text{as}} \text{CNot } (\text{mset } l - \{\#a\# \}) \text{ then } \text{Some } a \text{ else } \text{None}$   
 $| a \# ab \# xa \Rightarrow \text{Map.empty } xa) = \text{Some } a$

**thus**  $a \in \text{set } l$

**by** (case-tac  $[a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M]$ )

(fastforce dest: filter-eq-ConsD split: split-if-asm split: list.splits)+

**qed**

**lemma** *is-unit-clause-nil*[*simp*]:  $\text{is-unit-clause } []\ M = \text{None}$

**unfolding** *is-unit-clause-def* **by** auto

## 18.1.2 Unit propagation for all clauses

Finding the first clause to propagate

**fun** *find-first-unit-clause* :: 'a literal list list  $\Rightarrow$  ('a, 'b, 'c) marked-lit list  
 $\Rightarrow$  ('a literal  $\times$  'a literal list) option **where**

*find-first-unit-clause*  $(a \# l)\ M =$

(case *is-unit-clause*  $a\ M$  of

$\text{None} \Rightarrow \text{find-first-unit-clause } l\ M$

$| \text{Some } L \Rightarrow \text{Some } (L, a))$  |

*find-first-unit-clause*  $[]$  - = None

**lemma** *find-first-unit-clause-some*:  
*find-first-unit-clause*  $l$   $M = \text{Some } (a, c)$   
 $\implies c \in \text{set } l \wedge M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \}) \wedge \text{undefined-lit } M a \wedge a \in \text{set } c$   
**apply** (*induction*  $l$ )  
**apply** *simp*  
**by** (*auto split: option.splits dest: is-unit-clause-some-in is-unit-clause-some-CNot is-unit-clause-some-undef*)

**lemma** *propagate-is-unit-clause-not-None*:  
**assumes** *dist: distinct*  $c$  **and**  
 $M: M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \})$  **and**  
*undef: undefined-lit*  $M a$  **and**  
 $ac: a \in \text{set } c$   
**shows** *is-unit-clause*  $c$   $M \neq \text{None}$

**proof** –  
**have**  $[a \leftarrow c . \text{atm-of } a \notin \text{atm-of ' lits-of } M] = [a]$   
**using** *assms*  
**proof** (*induction*  $c$ )  
**case** *Nil* **thus** *?case* **by** *simp*  
**next**  
**case** (*Cons*  $ac$   $c$ )  
**show** *?case*  
**proof** (*cases*  $a = ac$ )  
**case** *True*  
**thus** *?thesis* **using** *Cons*  
**by** (*auto simp del: lits-of-unfold*  
*simp add: lits-of-unfold[symmetric] Marked-Propagated-in-iff-in-lits-of*  
*atm-of-eq-atm-of atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)  
**next**  
**case** *False*  
**hence**  $T: \text{mset } c + \{\#ac\# \} - \{\#a\# \} = \text{mset } c - \{\#a\# \} + \{\#ac\# \}$   
**by** (*auto simp add: multiset-eq-iff*)  
**show** *?thesis* **using** *False Cons*  
**by** (*auto simp add: T atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)  
**qed**  
**qed**  
**thus** *?thesis*  
**using**  $M$  **unfolding** *is-unit-clause-def* **by** *auto*  
**qed**

**lemma** *find-first-unit-clause-none*:  
 $\text{distinct } c \implies c \in \text{set } l \implies M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \}) \implies \text{undefined-lit } M a \implies a \in \text{set } c$   
 $\implies \text{find-first-unit-clause } l M \neq \text{None}$   
**by** (*induction*  $l$ )  
*(auto split: option.split simp add: propagate-is-unit-clause-not-None)*

### 18.1.3 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*  
 $\text{None} \Rightarrow \text{find-first-unused-var } l M$   
 $| \text{Some } a \Rightarrow \text{Some } a$   $|$   
*find-first-unused-var*  $[] - = \text{None}$

**lemma** *find-none[iff]*:

```

List.find (λlit. lit ∉ M ∧ ¬lit ∉ M) a = None ⟷ atm-of ' set a ⊆ atm-of ' M
apply (induct a)
using atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
by (force simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)+

lemma find-some: List.find (λlit. lit ∉ M ∧ ¬lit ∉ M) a = Some b ⟹ b ∈ set a ∧ b ∉ M ∧ ¬b ∉ M
unfolding find-Some-iff by (metis nth-mem)

lemma find-first-unused-var-None[iff]:
  find-first-unused-var l M = None ⟷ (∀ a ∈ set l. atm-of ' set a ⊆ atm-of ' M)
by (induct l)
  (auto split: option.splits dest!: find-some
    simp add: image-subset-iff atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)

lemma find-first-unused-var-Some-not-all-incl:
  assumes find-first-unused-var l M = Some c
  shows ¬(∀ a ∈ set l. atm-of ' set a ⊆ atm-of ' M)
proof -
  have find-first-unused-var l M ≠ None
  using assms by (cases find-first-unused-var l M) auto
  thus ¬(∀ a ∈ set l. atm-of ' set a ⊆ atm-of ' M) by auto
qed

lemma find-first-unused-var-Some:
  find-first-unused-var l M = Some a ⟹ (∃ m ∈ set l. a ∈ set m ∧ a ∉ M ∧ ¬a ∉ M)
by (induct l) (auto split: option.splits dest: find-some)

lemma find-first-unused-var-undefined:
  find-first-unused-var l (lits-of Ms) = Some a ⟹ undefined-lit Ms a
  using find-first-unused-var-Some[of l lits-of Ms a] Marked-Propagated-in-iff-in-lits-of
  by blast

end
theory DPLL-W-Implementation
imports DPLL-CDCL-W-Implementation DPLL-W ~~/src/HOL/Library/Code-Target-Numeral
begin

```

## 18.2 Simple Implementation of DPLL

### 18.2.1 Combining the propagate and decide: a DPLL step

**definition** *DPLL-step* :: int dpll<sub>W</sub>-marked-lits × int literal list list

⇒ int dpll<sub>W</sub>-marked-lits × int literal list list **where**

*DPLL-step* = (λ(Ms, N).

```

  (case find-first-unit-clause N Ms of
    Some (L, -) ⇒ (Propagated L () # Ms, N)
  | - ⇒
    if ∃ C ∈ set N. (∀ c ∈ set C. ¬c ∈ lits-of Ms)
    then
      (case backtrack-split Ms of
        (-, L # M) ⇒ (Propagated (- (lit-of L)) () # M, N)
      | (-, -) ⇒ (Ms, N)
      )
    else
      (case find-first-unused-var N (lits-of Ms) of
        Some a ⇒ (Marked a () # Ms, N)

```

| None  $\Rightarrow$  (Ms, N))))

Example of propagation:

**value** *DPLL-step* ([*Marked* (*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*  $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list})$   
 $(N:: \text{int literal list list}). (Ms, \text{mset} (\text{map mset } N))$

**abbreviation** *toS'*  $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list},$   
 $N:: \text{int literal list list}). (Ms, \text{mset} (\text{map mset } N))$

Proof of correctness of *DPLL-step*

**lemma** *DPLL-step-is-a-dpll<sub>W</sub>-step*:

**assumes** *step*: (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*)

**and** *neq*: (*Ms*, *N*)  $\neq$  (*Ms'*, *N'*)

**shows** *dpll<sub>W</sub>* (*toS Ms N*) (*toS Ms' N'*)

**proof** –

**let** ?*S* = (*Ms*, *mset* (*map mset N*))

**{ fix** *L E*

**assume** *unit*: *find-first-unit-clause N Ms* = *Some* (*L*, *E*)

**hence** *Ms'N*: (*Ms'*, *N'*) = (*Propagated L* () # *Ms*, *N*)

**using** *step* **unfolding** *DPLL-step-def* **by** *auto*

**obtain** *C* **where**

*C*: *C*  $\in$  *set N* **and**

*Ms*: *Ms*  $\models_{\text{as}}$  *CNot* (*mset C* – {#*L*#}) **and**

*undef*: *undefined-lit Ms L* **and**

*L*  $\in$  *set C* **using** *find-first-unit-clause-some[OF unit]* **by** *metis*

**have** *dpll<sub>W</sub>* (*Ms*, *mset* (*map mset N*))

(*Propagated L* () # *fst* (*Ms*, *mset* (*map mset N*)), *snd* (*Ms*, *mset* (*map mset N*)))

**apply** (*rule dpll<sub>W</sub>.propagate*)

**using** *Ms undef C* (*L*  $\in$  *set C*) **unfolding** *mem-set-multiset-eq* **by** (*auto simp add: C*)

**hence** ?*thesis* **using** *Ms'N* **by** *auto*

**}**

**moreover**

**{ assume** *unit*: *find-first-unit-clause N Ms* = *None*

**assume** *exC*:  $\exists C \in \text{set } N. Ms \models_{\text{as}} CNot (\text{mset } C)$

**then obtain** *C* **where** *C*: *C*  $\in$  *set N* **and** *Ms*: *Ms*  $\models_{\text{as}}$  *CNot* (*mset C*) **by** *auto*

**then obtain** *L M M'* **where** *bt*: *backtrack-split Ms* = (*M'*, *L* # *M*)

**using** *step exC neq* **unfolding** *DPLL-step-def prod.case unit*

**by** (*cases backtrack-split Ms*, *case-tac b*) *auto*

**hence** *is-marked L* **using** *backtrack-split-snd-hd-marked[of Ms]* **by** *auto*

**have** 1: *dpll<sub>W</sub>* (*Ms*, *mset* (*map mset N*))

(*Propagated* (– *lit-of L*) () # *M*, *snd* (*Ms*, *mset* (*map mset N*)))

**apply** (*rule dpll<sub>W</sub>.backtrack[OF - is-marked L, of ]*)

**using** *C Ms bt* **by** *auto*

**moreover have** (*Ms'*, *N'*) = (*Propagated* (– (*lit-of L*)) () # *M*, *N*)

**using** *step exC* **unfolding** *DPLL-step-def bt prod.case unit* **by** *auto*

**ultimately have** ?*thesis* **by** *auto*

**}**

**moreover**

**{ assume** *unit*: *find-first-unit-clause N Ms* = *None*

**assume** *exC*:  $\neg (\exists C \in \text{set } N. Ms \models_{\text{as}} CNot (\text{mset } C))$

**obtain** *L* **where** *unused*: *find-first-unused-var N (lits-of Ms)* = *Some L*

**using** *step exC neq* **unfolding** *DPLL-step-def prod.case unit*

```

  by (cases find-first-unused-var N (lits-of Ms)) auto
have dpllW (Ms, mset (map mset N))
  (Marked L () # fst (Ms, mset (map mset N)), snd (Ms, mset (map mset N)))
  apply (rule dpllW.decided[of ?S L])
  using find-first-unused-var-Some[OF unused]
  by (auto simp add: Marked-Propagated-in-iff-in-lits-of atms-of-m-def)
moreover have (Ms', N') = (Marked L () # Ms, N)
  using step exC unfolding DPLL-step-def unused prod.case unit by auto
ultimately have ?thesis by auto
}
ultimately show ?thesis by (cases find-first-unit-clause N Ms) auto
qed

lemma DPLL-step-stuck-final-state:
  assumes step: (Ms, N) = DPLL-step (Ms, N)
  shows conclusive-dpllW-state (toS Ms N)
proof -
  have unit: find-first-unit-clause N Ms = None
    using step unfolding DPLL-step-def by (auto split: option.splits)

  { assume n:  $\exists C \in \text{set } N. Ms \models_{as} CNot (mset C)$ 
    hence Ms: (Ms, N) = (case backtrack-split Ms of (x, [])  $\Rightarrow$  (Ms, N)
      | (x, L # M)  $\Rightarrow$  (Propagated (- lit-of L) () # M, N))
      using step unfolding DPLL-step-def by (simp add: unit)
  }

  have snd (backtrack-split Ms) = []
  proof (cases backtrack-split Ms, cases snd (backtrack-split Ms))
    fix a b
    assume backtrack-split Ms = (a, b) and snd (backtrack-split Ms) = []
    thus snd (backtrack-split Ms) = [] by blast
  next
    fix a b aa list
    assume
      bt: backtrack-split Ms = (a, b) and
      bt': snd (backtrack-split Ms) = aa # list
    hence Ms: Ms = Propagated (- lit-of aa) () # list using Ms by auto
    have is-marked aa using backtrack-split-snd-hd-marked[of Ms] bt bt' by auto
    moreover have fst (backtrack-split Ms) @ aa # list = Ms
      using backtrack-split-list-eq[of Ms] bt' by auto
    ultimately have False unfolding Ms by auto
    thus snd (backtrack-split Ms) = [] by blast
  qed

  hence ?thesis
    using n backtrack-snd-empty-not-marked[of Ms] unfolding conclusive-dpllW-state-def
    by (cases backtrack-split Ms) auto
}
moreover {
  assume n:  $\neg (\exists C \in \text{set } N. Ms \models_{as} CNot (mset C))$ 
  hence find-first-unused-var N (lits-of Ms) = None
    using step unfolding DPLL-step-def by (simp add: unit split: option.splits)
  hence a:  $\forall a \in \text{set } N. \text{atm-of } a \subseteq \text{atm-of } (lits-of Ms)$  by auto
  have fst (toS Ms N)  $\models_{asm}$  snd (toS Ms N) unfolding true-annots-def CNot-def Ball-def
  proof clarify
    fix x

```



```

    assume  $x: x \in \text{set-mset } (\text{clauses } (\text{toS } Ms \ N))$ 
    hence  $\neg Ms \models_{as} CNot \ x$  using  $n$  unfolding  $\text{true-annot-def } CNot\text{-def } Ball\text{-def}$  by  $\text{auto}$ 
    moreover have  $\text{total-over-m } (\text{lits-of } Ms) \ \{x\}$ 
      using  $a \ x \ \text{image-iff in-mono } \text{atms-of-s-def}$ 
      unfolding  $\text{total-over-m-def } \text{total-over-set-def } \text{lits-of-def}$  by  $\text{fastforce}$ 
    ultimately show  $\text{fst } (\text{toS } Ms \ N) \models_a x$ 
      using  $\text{total-not-CNot[of lits-of } Ms \ x]$  by  $(\text{simp add: true-annot-def true-annot-def true-cl})$ 
  qed
  hence  $?thesis$  unfolding  $\text{conclusive-dpll}_W\text{-state-def}$  by  $\text{blast}$ 
}
ultimately show  $?thesis$  by  $\text{blast}$ 
qed

```

### 18.2.2 Adding invariants

**Invariant tested in the function**  $\text{function } DPLL\text{-ci} :: \text{int } dpll_W\text{-marked-lits} \Rightarrow \text{int literal list list}$

$\Rightarrow \text{int } dpll_W\text{-marked-lits} \times \text{int literal list list}$  **where**

```

DPLL-ci Ms N =
  (if  $\neg dpll_W\text{-all-inv } (Ms, \text{mset } (\text{map mset } N))$ 
   then  $(Ms, N)$ 
   else
    let  $(Ms', N') = DPLL\text{-step } (Ms, N)$  in
    if  $(Ms', N') = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-ci } Ms' \ N$ )
by fast+

```

**termination**

**proof**  $(\text{relation } \{(S', S). (\text{toS}' S', \text{toS}' S) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}\})$

**show**  $\text{wf } \{(S', S). (\text{toS}' S', \text{toS}' S) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}\}$

**using**  $\text{wf-if-measure-f}[OF \ dpll_W\text{-wf, of toS}']$  **by**  $\text{auto}$

**next**

**fix**  $Ms :: \text{int } dpll_W\text{-marked-lits}$  **and**  $N \ x \ xa \ y$

**assume**  $\neg \neg dpll_W\text{-all-inv } (\text{toS } Ms \ N)$

**and**  $\text{step}: x = DPLL\text{-step } (Ms, N)$

**and**  $x: (xa, y) = x$

**and**  $(xa, y) \neq (Ms, N)$

**thus**  $((xa, N), Ms, N) \in \{(S', S). (\text{toS}' S', \text{toS}' S) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}\}$

**using**  $DPLL\text{-step-is-a-dpll}_W\text{-step } dpll_W\text{-same-clauses } \text{split-conv}$  **by**  $\text{fastforce}$

**qed**

**No invariant tested**  $\text{function } (\text{domintros}) \ DPLL\text{-part} :: \text{int } dpll_W\text{-marked-lits} \Rightarrow \text{int literal list list}$

$\Rightarrow$

$\text{int } dpll_W\text{-marked-lits} \times \text{int literal list list}$  **where**

$DPLL\text{-part } Ms \ N =$

$(\text{let } (Ms', N') = DPLL\text{-step } (Ms, N)$  in

$\text{if } (Ms', N') = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-part } Ms' \ N$ )

**by fast+**

**lemma**  $\text{snd-DPLL-step[simp]}:$

$\text{snd } (DPLL\text{-step } (Ms, N)) = N$

**unfolding**  $DPLL\text{-step-def}$  **by**  $(\text{auto split: split-if option.splits prod.splits list.splits})$

**lemma**  $dpll_W\text{-all-inv-implicS-2-eq3-and-dom}:$

**assumes**  $dpll_W\text{-all-inv } (Ms, \text{mset } (\text{map mset } N))$

**shows**  $DPLL\text{-ci } Ms \ N = DPLL\text{-part } Ms \ N \wedge DPLL\text{-part-dom } (Ms, N)$

**using**  $\text{assms}$

**proof**  $(\text{induct rule: } DPLL\text{-ci.induct})$

```

case (1 Ms N)
have snd (DPLL-step (Ms, N)) = N by auto
then obtain Ms' where Ms': DPLL-step (Ms, N) = (Ms', N) by (case-tac DPLL-step (Ms, N)) auto
have inv': dpllW-all-inv (toS Ms' N) by (metis (mono-tags) 1.prem DPLL-step-is-a-dpllW-step Ms'
  dpllW-all-inv old.prod.inject)
{ assume (Ms', N) ≠ (Ms, N)
  hence DPLL-ci Ms' N = DPLL-part Ms' N ∧ DPLL-part-dom (Ms', N) using 1(1)[of - Ms' N]
Ms'
  1(2) inv' by auto
  hence DPLL-part-dom (Ms, N) using DPLL-part.domintros Ms' by fastforce
  moreover have DPLL-ci Ms N = DPLL-part Ms N using 1.prem DPLL-part.psimps Ms'
    ⟨DPLL-ci Ms' N = DPLL-part Ms' N ∧ DPLL-part-dom (Ms', N)⟩ ⟨DPLL-part-dom (Ms, N)⟩ by
auto
    ultimately have ?case by blast
  }
moreover {
  assume (Ms', N) = (Ms, N)
  hence ?case using DPLL-part.domintros DPLL-part.psimps Ms' by fastforce
  }
ultimately show ?case by blast
qed

lemma DPLL-ci-dpllW-rtrancp:
  assumes DPLL-ci Ms N = (Ms', N')
  shows dpllW** (toS Ms N) (toS Ms' N)
  using assms
proof (induct Ms N arbitrary: Ms' N' rule: DPLL-ci.induct)
  case (1 Ms N Ms' N') note IH = this(1) and step = this(2)
  obtain S1 S2 where S: (S1, S2) = DPLL-step (Ms, N) by (case-tac DPLL-step (Ms, N)) auto

  { assume ¬dpllW-all-inv (toS Ms N)
    hence (Ms, N) = (Ms', N) using step by auto
    hence ?case by auto
  }
  moreover
  { assume dpllW-all-inv (toS Ms N)
    and (S1, S2) = (Ms, N)
    hence ?case using S step by auto
  }
  moreover
  { assume dpllW-all-inv (toS Ms N)
    and (S1, S2) ≠ (Ms, N)
    moreover obtain S1' S2' where DPLL-ci S1 N = (S1', S2') by (case-tac DPLL-ci S1 N) auto
    moreover have DPLL-ci Ms N = DPLL-ci S1 N using DPLL-ci.simps[of Ms N] calculation
    proof -
      have (case (S1, S2) of (ms, lss) ⇒
        if (ms, lss) = (Ms, N) then (Ms, N) else DPLL-ci ms N = DPLL-ci Ms N
        using S DPLL-ci.simps[of Ms N] calculation by presburger
        hence (if (S1, S2) = (Ms, N) then (Ms, N) else DPLL-ci S1 N = DPLL-ci Ms N
        by fastforce
        thus ?thesis
        using calculation(2) by presburger
      )
    qed
    ultimately have dpllW** (toS S1' N) (toS Ms' N) using IH[of (S1, S2) S1 S2] S step by simp
  }

```

```

moreover have  $dpll_W$  (toS Ms N) (toS S1 N)
  by (metis DPLL-step-is-a-dpllW-step S  $\langle(S_1, S_2) \neq (Ms, N)\rangle$  prod.sel(2) snd-DPLL-step)
ultimately have ?case by (metis (mono-tags, hide-lams) IH S  $\langle(S_1, S_2) \neq (Ms, N)\rangle$ 
   $\langle DPLL-ci\ Ms\ N = DPLL-ci\ S_1\ N \rangle$   $dpll_W$ -all-inv (toS Ms N) converse-rtrancpl-into-rtrancpl
  local.step)
}
ultimately show ?case by blast
qed

```

**lemma**  $dpll_W$ -all-inv- $dpll_W$ -trancpl-irrefl:

```

assumes  $dpll_W$ -all-inv (Ms, N)
and  $dpll_W^{++}$  (Ms, N) (Ms, N)
shows False

```

**proof** –

```

have 1: wf  $\{(S', S). dpll_W$ -all-inv S  $\wedge dpll_W^{++}$  S S' $\}$  using  $dpll_W$ -wf-trancpl by auto
have ((Ms, N), (Ms, N))  $\in \{(S', S). dpll_W$ -all-inv S  $\wedge dpll_W^{++}$  S S' $\}$  using assms by auto
thus False using wf-not-refl[OF 1] by blast

```

**qed**

**lemma** DPLL-ci-final-state:

```

assumes step: DPLL-ci Ms N = (Ms, N)
and inv:  $dpll_W$ -all-inv (toS Ms N)
shows conclusive- $dpll_W$ -state (toS Ms N)

```

**proof** –

```

have st:  $dpll_W^{**}$  (toS Ms N) (toS Ms N) using DPLL-ci- $dpll_W$ -rtrancpl[OF step] .
have DPLL-step (Ms, N) = (Ms, N)

```

**proof** (rule ccontr)

```

obtain Ms' N' where Ms'N: (Ms', N') = DPLL-step (Ms, N)

```

```

by (case-tac DPLL-step (Ms, N)) auto

```

```

assume  $\neg$  ?thesis

```

```

hence DPLL-ci Ms' N = (Ms, N) using step inv st Ms'N[symmetric] by fastforce

```

```

hence  $dpll_W^{++}$  (toS Ms N) (toS Ms N)

```

```

by (metis DPLL-ci- $dpll_W$ -rtrancpl DPLL-step-is-a- $dpll_W$ -step Ms'N  $\langle DPLL-step\ (Ms, N) \neq (Ms,$ 
N) $\rangle$ 

```

```

  prod.sel(2) rtrancpl-into-trancpl2 snd-DPLL-step)

```

```

thus False using  $dpll_W$ -all-inv- $dpll_W$ -trancpl-irrefl inv by auto

```

**qed**

```

thus ?thesis using DPLL-step-stuck-final-state[of Ms N] by simp

```

**qed**

**lemma** DPLL-step-obtains:

```

obtains Ms' where (Ms', N) = DPLL-step (Ms, N)

```

```

unfolding DPLL-step-def by (metis (no-types, lifting) DPLL-step-def prod.collapse snd-DPLL-step)

```

**lemma** DPLL-ci-obtains:

```

obtains Ms' where (Ms', N) = DPLL-ci Ms N

```

**proof** (induct rule: DPLL-ci.induct)

```

case (1 Ms N) note IH = this(1) and that = this(2)

```

```

obtain S where SN: (S, N) = DPLL-step (Ms, N) using DPLL-step-obtains by metis

```

```

{ assume  $\neg dpll_W$ -all-inv (toS Ms N)

```

```

  hence ?case using that by auto

```

```

}

```

```

moreover {

```

```

  assume n: (S, N)  $\neq$  (Ms, N)

```

```

  and inv:  $dpll_W$ -all-inv (toS Ms N)

```

```

have  $\exists ms. DPLL\text{-}step (Ms, N) = (ms, N)$ 
  by (metis  $\langle \bigwedge thesisa. (\bigwedge S. (S, N) = DPLL\text{-}step (Ms, N) \implies thesisa) \implies thesisa \rangle$ )
hence ?thesis
  using IH that by fastforce
}
moreover {
  assume  $n: (S, N) = (Ms, N)$ 
  hence ?case using SN that by fastforce
}
ultimately show ?case by blast
qed

```

**lemma** *DPLL-ci-no-more-step:*

```

assumes step:  $DPLL\text{-}ci\ Ms\ N = (Ms', N')$ 
shows  $DPLL\text{-}ci\ Ms'\ N' = (Ms', N')$ 
using assms
proof (induct arbitrary:  $Ms'\ N'$  rule:  $DPLL\text{-}ci.induct$ )
case (1  $Ms\ N\ Ms'\ N'$ ) note IH = this(1) and step = this(2)
obtain  $S_1$  where  $S: (S_1, N) = DPLL\text{-}step (Ms, N)$  using  $DPLL\text{-}step\text{-}obtains$  by auto
{ assume  $\neg dpll_W\text{-}all\text{-}inv (toS\ Ms\ N)$ 
  hence ?case using step by auto
}
moreover {
  assume  $dpll_W\text{-}all\text{-}inv (toS\ Ms\ N)$ 
  and  $(S_1, N) = (Ms, N)$ 
  hence ?case using S step by auto
}
moreover
{ assume  $inv: dpll_W\text{-}all\text{-}inv (toS\ Ms\ N)$ 
  assume  $n: (S_1, N) \neq (Ms, N)$ 
  obtain  $S_1'$  where  $SS: (S_1', N) = DPLL\text{-}ci\ S_1\ N$  using  $DPLL\text{-}ci\text{-}obtains$  by blast
  moreover have  $DPLL\text{-}ci\ Ms\ N = DPLL\text{-}ci\ S_1\ N$ 
  proof -
    have (case  $(S_1, N)$  of  $(ms, lss) \Rightarrow$  if  $(ms, lss) = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-}ci\ ms\ N$ )
      =  $DPLL\text{-}ci\ Ms\ N$ 
    using S  $DPLL\text{-}ci.simps[of\ Ms\ N]$  calculation inv by presburger
    hence (if  $(S_1, N) = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-}ci\ S_1\ N$ ) =  $DPLL\text{-}ci\ Ms\ N$ 
    by fastforce
    thus ?thesis
    using calculation n by presburger
  qed
  moreover
    have  $DPLL\text{-}ci\ S_1'\ N = (S_1', N)$  using step IH[OF - - S n SS[symmetric]] inv by blast
  ultimately have ?case using step by fastforce
}
ultimately show ?case by blast
qed

```

**lemma** *DPLL-part-dpll<sub>W</sub>-all-inv-final:*

```

fixes  $M\ Ms':: (int, unit, unit)$  marked-lit list and
   $N:: int$  literal list list
assumes inv:  $dpll_W\text{-}all\text{-}inv (Ms, mset (map mset N))$ 
and  $MsN: DPLL\text{-}part\ Ms\ N = (Ms', N)$ 

```

**shows** *conclusive-dpll<sub>W</sub>-state* (toS Ms' N)  $\wedge$  *dpll<sub>W</sub>\*\** (toS Ms N) (toS Ms' N)  
**proof** –  
**have** 2: *DPLL-ci* Ms N = *DPLL-part* Ms N **using** *inv dpll<sub>W</sub>-all-inv-implieS-2-eq3-and-dom* **by** *blast*  
**hence** *star*: *dpll<sub>W</sub>\*\** (toS Ms N) (toS Ms' N) **unfolding** *MsN* **using** *DPLL-ci-dpll<sub>W</sub>-rtrancp* **by**  
*blast*  
**hence** *inv'*: *dpll<sub>W</sub>-all-inv* (toS Ms' N) **using** *inv rtrancp-dpll<sub>W</sub>-all-inv* **by** *blast*  
**show** ?thesis **using** *star DPLL-ci-final-state[OF DPLL-ci-no-more-step inv']* 2 **unfolding** *MsN* **by**  
*blast*  
**qed**

## Embedding the invariant into the type

**Defining the type** **typedef** *dpll<sub>W</sub>-state* =

$\{(M::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list}, N::\text{int literal list list}).$   
 $\text{dpll}_W\text{-all-inv (toS M N)}\}$

**morphisms** *rough-state-of state-of*

**proof**

**show**  $([], []) \in \{(M, N). \text{dpll}_W\text{-all-inv (toS M N)}\}$  **by** (*auto simp add: dpll<sub>W</sub>-all-inv-def*)

**qed**

**lemma**

*DPLL-part-dom*  $([], N)$

**using** *assms dpll<sub>W</sub>-all-inv-implieS-2-eq3-and-dom*[of  $[], N$ ] **by** (*simp add: dpll<sub>W</sub>-all-inv-def*)

**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'* = (*rough-state-of S* = *rough-state-of S'*)

**instance**

**by** *standard (simp add: rough-state-of-inject equal-dpll<sub>W</sub>-state-def)*

**end**

**DPLL** **definition** *DPLL-step'* :: *dpll<sub>W</sub>-state*  $\Rightarrow$  *dpll<sub>W</sub>-state* **where**

*DPLL-step'* *S* = *state-of (DPLL-step (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)*  $\in \{(M, N). \text{dpll}_W\text{-all-inv (toS M N)}\}$

**by** (*smt DPLL-ci.simps DPLL-ci-dpll<sub>W</sub>-rtrancp case-prodE case-prodI2 rough-state-of mem-Collect-eq old.prod.case prod.sel(2) rtrancp-dpll<sub>W</sub>-all-inv snd-DPLL-step*)

**lemma** *rough-state-of-DPLL-step'-DPLL-step*[*simp*]:

*rough-state-of (DPLL-step' S)* = *DPLL-step (rough-state-of S)*

**using** *DPLL-step-dpll<sub>W</sub>-conc-inv DPLL-step'-def state-of-inverse* **by** *auto*

**function** *DPLL-tot*:: *dpll<sub>W</sub>-state*  $\Rightarrow$  *dpll<sub>W</sub>-state* **where**

*DPLL-tot S* =

(*let S' = DPLL-step' S* in

*if S' = S* then *S* else *DPLL-tot S'*)

**by** *fast+*

**termination**

**proof** (*relation*  $\{(T', T).$

*(rough-state-of T', rough-state-of T)*

$\in \{(S', S). (\text{toS}' S', \text{toS}' S)\}$

```

      ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}))
show wf {(b, a).
  (rough-state-of b, rough-state-of a)
  ∈ {(b, a). (toS' b, toS' a)
    ∈ {(b, a). dpllW-all-inv a ∧ dpllW a b}}}}
  using wf-if-measure-f[OF wf-if-measure-f[OF dpllW-wf, of toS'], of rough-state-of] .
next
fix S x
assume x: x = DPLL-step' S
and x ≠ S
have dpllW-all-inv (case rough-state-of S of (Ms, N) ⇒ (Ms, mset (map mset N)))
  by (metis (no-types, lifting) case-prodE mem-Collect-eq old.prod.case rough-state-of)
moreover have dpllW (case rough-state-of S of (Ms, N) ⇒ (Ms, mset (map mset N)))
  (case rough-state-of (DPLL-step' S) of (Ms, N) ⇒ (Ms, mset (map mset N)))
proof -
  obtain Ms N where Ms: (Ms, N) = rough-state-of S by (cases rough-state-of S) auto
  have dpllW-all-inv (toS' (Ms, N)) using calculation unfolding Ms by blast
  moreover obtain Ms' N' where Ms': (Ms', N') = rough-state-of (DPLL-step' S)
    by (cases rough-state-of (DPLL-step' S)) auto
  ultimately have dpllW-all-inv (toS' (Ms', N')) unfolding Ms'
    by (metis (no-types, lifting) case-prod-unfold mem-Collect-eq rough-state-of)

  have dpllW (toS Ms N) (toS Ms' N')
    apply (rule DPLL-step-is-a-dpllW-step[of Ms' N' Ms N])
    unfolding Ms Ms' using ⟨x ≠ S⟩ rough-state-of-inject x by fastforce+
    thus ?thesis unfolding Ms[symmetric] Ms'[symmetric] by auto
qed
ultimately show (x, S) ∈ {(T', T). (rough-state-of T', rough-state-of T)
  ∈ {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}}
  by (auto simp add: x)
qed

lemma [code]:
DPLL-tot S =
  (let S' = DPLL-step' S in
  if S' = S then S else DPLL-tot S') by auto

lemma DPLL-tot-DPLL-step-DPLL-tot[simp]: DPLL-tot (DPLL-step' S) = DPLL-tot S
  apply (cases DPLL-step' S = S)
  apply simp
  unfolding DPLL-tot.simps[of S] by (simp del: DPLL-tot.simps)

lemma DOPLL-step'-DPLL-tot[simp]:
  DPLL-step' (DPLL-tot S) = DPLL-tot S
  by (rule DPLL-tot.induct[of λS. DPLL-step' (DPLL-tot S) = DPLL-tot S S])
  (metis (full-types) DPLL-tot.simps)

lemma DPLL-tot-final-state:
  assumes DPLL-tot S = S
  shows conclusive-dpllW-state (toS' (rough-state-of S))
proof -
  have DPLL-step' S = S using assms[symmetric] DOPLL-step'-DPLL-tot by metis
  hence DPLL-step (rough-state-of S) = (rough-state-of S)

```

**unfolding** *DPLL-step'-def* **using** *DPLL-step-dpll<sub>W</sub>-conc-inv rough-state-of-inverse*  
**by** (*metis rough-state-of-DPLL-step'-DPLL-step*)  
**thus** *?thesis*  
**by** (*metis (mono-tags, lifting) DPLL-step-stuck-final-state old.prod.exhaust split-conv*)  
**qed**

**lemma** *DPLL-tot-star*:

**assumes** *rough-state-of (DPLL-tot S) = S'*  
**shows** *dpll<sub>W</sub>\*\* (toS' (rough-state-of S)) (toS' S')*  
**using** *assms*  
**proof** (*induction arbitrary: S' rule: DPLL-tot.induct*)  
**case** (1 *S S'*)  
**let** *?x = DPLL-step' S*  
**{ assume** *?x = S*  
**then have** *?case* **using** 1(2) **by** *simp*  
**}**  
**moreover {**  
**assume** *S: ?x ≠ S*  
**have** *?case*  
**apply** (*cases DPLL-step' S = S*)  
**using** *S* **apply** *blast*  
**by** (*smt 1.IH 1.prem DPLL-step-is-a-dpll<sub>W</sub>-step DPLL-tot.simps case-prodE2*  
*rough-state-of-DPLL-step'-DPLL-step rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl*  
*rtranclp-idemp split-conv*)  
**}**  
**ultimately show** *?case* **by** *auto*  
**qed**

**lemma** *rough-state-of-rough-state-of-nil[simp]*:

*rough-state-of (state-of ([], N)) = ([], N)*  
**apply** (*rule DPLL-W-Implementation.dpll<sub>W</sub>-state.state-of-inverse*)  
**unfolding** *dpll<sub>W</sub>-all-inv-def* **by** *auto*

Theorem of correctness

**lemma** *DPLL-tot-correct*:

**assumes** *rough-state-of (DPLL-tot (state-of ([], N))) = (M, N')*  
**and** *(M', N'') = toS' (M, N')*  
**shows** *M' ⊨<sub>asm</sub> N'' ↔ satisfiable (set-mset N'')*  
**proof** –  
**have** *dpll<sub>W</sub>\*\* (toS' ([], N)) (toS' (M, N'))* **using** *DPLL-tot-star[OF assms(1)]* **by** *auto*  
**moreover have** *conclusive-dpll<sub>W</sub>-state (toS' (M, N'))*  
**using** *DPLL-tot-final-state* **by** (*metis (mono-tags, lifting) DPLL-step'-DPLL-tot DPLL-tot.simps*  
*assms(1)*)  
**ultimately show** *?thesis* **using** *dpll<sub>W</sub>-conclusive-state-correct* **by** (*smt DPLL-ci.simps*  
*DPLL-ci-dpll<sub>W</sub>-rtranclp assms(2) dpll<sub>W</sub>-all-inv-def prod.case prod.sel(1) prod.sel(2)*  
*rtranclp-dpll<sub>W</sub>-inv(3) rtranclp-dpll<sub>W</sub>-inv-starting-from-0*)  
**qed**

### 18.2.3 Code export

**A conversion to DPLL-W-Implementation.dpll<sub>W</sub>-state** **definition** *Con :: (int, unit, unit) marked-lit list × 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*

**using** *rough-state-of*[*of S*] **unfolding** *Con-def* **by** *auto*

**declare** *rough-state-of-DPLL-step'-DPLL-step*[*code abstract*]

**lemma** *Con-DPLL-step-rough-state-of-state-of[simp]*:

*Con (DPLL-step (rough-state-of s)) = state-of (DPLL-step (rough-state-of s))*

**unfolding** *Con-def* **by** (*metis* (*mono-tags*, *lifting*) *DPLL-step-dpll<sub>W</sub>-conc-inv mem-Collect-eq prod.case-eq-if*)

A slightly different version of *DPLL-tot* where the returned boolean indicates the result.

**definition** *DPLL-tot-rep* **where**

*DPLL-tot-rep S =*

*(let (M, N) = (rough-state-of (DPLL-tot S)) in (∀ A ∈ set N. (∃ a ∈ set A. a ∈ lits-of (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**

**notation** *image-mset* (**infixr** *'#* 90)

**type-synonym** *'a cdcl<sub>W</sub>-mark* = *'a clause*

**type-synonym** *cdcl<sub>W</sub>-marked-level* = *nat*

**type-synonym** *'v cdcl<sub>W</sub>-marked-lit* = (*'v*, *cdcl<sub>W</sub>-marked-level*, *'v cdcl<sub>W</sub>-mark*) *marked-lit*

**type-synonym** *'v cdcl<sub>W</sub>-marked-lits* = (*'v*, *cdcl<sub>W</sub>-marked-level*, *'v cdcl<sub>W</sub>-mark*) *marked-lits*

**type-synonym** *'v cdcl<sub>W</sub>-state* =

*'v cdcl<sub>W</sub>-marked-lits × 'v clauses × 'v clauses × nat × 'v clause conflicting-clause*

**abbreviation** *trail* :: *'a × 'b × 'c × 'd × 'e ⇒ 'a* **where**

*trail ≡ (λ(M, -). M)*

**abbreviation** *cons-trail* :: *'a ⇒ 'a list × 'b × 'c × 'd × 'e ⇒ 'a list × 'b × 'c × 'd × 'e* **where**

*cons-trail ≡ (λL (M, S). (L#M, S))*

**abbreviation** *tl-trail* :: *'a list × 'b × 'c × 'd × 'e ⇒ 'a list × 'b × 'c × 'd × 'e* **where**

*tl-trail ≡ (λ(M, S). (tl M, S))*

**abbreviation** *clauses* :: *'a × 'b × 'c × 'd × 'e ⇒ 'b* **where**

*clauses ≡ λ(M, N, -). N*

**abbreviation** *learned-clss* :: *'a × 'b × 'c × 'd × 'e ⇒ 'c* **where**

*learned-clss ≡ λ(M, N, U, -). U*

**abbreviation** *backtrack-lvl* :: *'a × 'b × 'c × 'd × 'e ⇒ 'd* **where**



*backtrack-lvl*  $\equiv \lambda(M, N, U, k, -). k$

**abbreviation** *update-backtrack-lvl*  $:: 'd \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$   
**where**

*update-backtrack-lvl*  $\equiv \lambda k (M, N, U, -, S). (M, N, U, k, S)$

**abbreviation** *conflicting*  $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$  **where**

*conflicting*  $\equiv \lambda(M, N, U, k, D). D$

**abbreviation** *update-conflicting*  $:: 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$   
**where**

*update-conflicting*  $\equiv \lambda S (M, N, U, k, -). (M, N, U, k, S)$

**abbreviation** *S0-cdcl<sub>W</sub>*  $N \equiv (([], N, \{\#\}, 0, C\text{-True})). 'v \text{ cdcl}_W\text{-state}$

**abbreviation** *add-learned-cl*s **where**

*add-learned-cl*s  $\equiv \lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

**abbreviation** *remove-cl*s **where**

*remove-cl*s  $\equiv \lambda C (M, N, U, S). (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$

**interpretation** *cdcl<sub>W</sub>*: *state<sub>W</sub> trail clauses learned-clss backtrack-lvl conflicting*

$\lambda L (M, S). (L \# M, S)$

$\lambda(M, S). (tl \ M, S)$

$\lambda C (M, N, S). (M, \{\#C\# \} + N, S)$

$\lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

$\lambda C (M, N, U, S). (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$

$\lambda(k::nat) (M, N, U, -, D). (M, N, U, k, D)$

$\lambda D (M, N, U, k, -). (M, N, U, k, D)$

$\lambda N. ([], N, \{\#\}, 0, C\text{-True})$

$\lambda(-, N, U, -). ([], N, U, 0, C\text{-True})$

**by** *unfold-locales auto*

**lemma** *trail-conv*: *trail*  $(M, N, U, k, D) = M$  **and**

*clauses-conv*: *clauses*  $(M, N, U, k, D) = N$  **and**

*learned-clss-conv*: *learned-clss*  $(M, N, U, k, D) = U$  **and**

*conflicting-conv*: *conflicting*  $(M, N, U, k, D) = D$  **and**

*backtrack-lvl-conv*: *backtrack-lvl*  $(M, N, U, k, D) = k$

**by** *auto*

**lemma** *state-conv*:

$S = (\text{trail } S, \text{clauses } S, \text{learned-clss } S, \text{backtrack-lvl } S, \text{conflicting } S)$

**by** *(cases S) auto*

**interpretation** *cdcl<sub>W</sub>-termination* *trail clauses learned-clss backtrack-lvl conflicting*

$\lambda L (M, S). (L \# M, S)$

$\lambda(M, S). (tl \ M, S)$

$\lambda C (M, N, S). (M, \{\#C\# \} + N, S)$

$\lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

$\lambda C (M, N, U, S). (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$

$\lambda(k::nat) (M, N, U, -, D). (M, N, U, k, D)$

$\lambda D (M, N, U, k, -). (M, N, U, k, D)$

$\lambda N. ([], N, \{\#\}, 0, C\text{-True})$

$\lambda(-, N, U, -). ([], N, U, 0, C\text{-True})$

**by** *intro-locales*

**lemmas** *cdcl<sub>W</sub>.clauses-def[simp]*

```

lemma cdclW.state-eq-equality[iff]: cdclW.state-eq S T  $\longleftrightarrow$  S = T
  unfolding cdclW.state-eq-def by (cases S, cases T) auto
declare cdclW.state-simp[simp del]

```

## 18.3 CDCL Implementation

### 18.3.1 Definition of the rules

```

Types lemma true-clss-remdups[simp]:
  I  $\models_s$  (mset  $\circ$  remdups) ' N  $\longleftrightarrow$  I  $\models_s$  mset ' N
  by (simp add: true-clss-def)

```

```

lemma satisfiable-mset-remdups[simp]:
  satisfiable ((mset  $\circ$  remdups) ' N)  $\longleftrightarrow$  satisfiable (mset ' N)
unfolding satisfiable-carac[symmetric] by simp

```

```

declare mset-map[symmetric, simp]

```

```

value backtrack-split [Marked (Pos (Suc 0)) Level]
value  $\exists C \in \text{set } [[\text{Pos (Suc 0)}, \text{Neg (Suc 0)}]]$ . ( $\forall c \in \text{set } C$ .  $-c \in \text{lits-of } [\text{Marked (Pos (Suc 0)) Level}]$ )

```

```

type-synonym cdclW.state-inv-st = (nat, nat, nat literal list) marked-lit list  $\times$  nat literal list list
   $\times$  nat literal list list  $\times$  nat  $\times$  nat literal list conflicting-clause

```

We need some functions to convert between our abstract state *nat cdcl<sub>W</sub>.state* and the concrete state *cdcl<sub>W</sub>.state-inv-st*.

```

fun convert :: ('a, 'b, 'c list) marked-lit  $\Rightarrow$  ('a, 'b, 'c multiset) marked-lit where
  convert (Propagated L C) = Propagated L (mset C) |
  convert (Marked K i) = Marked K i

```

```

fun convertC :: 'a list conflicting-clause  $\Rightarrow$  'a multiset conflicting-clause where
  convertC (C-Clause C) = C-Clause (mset C) |
  convertC C-True = C-True

```

```

lemma convert-CTrue[iff]:
  convertC e = C-True  $\longleftrightarrow$  e = C-True
  by (cases e) auto

```

```

lemma convert-Propagated[elim!]:
  convert z = Propagated L C  $\Longrightarrow$  ( $\exists C'$ . z = Propagated L C'  $\wedge$  C = mset C')
  by (cases z) auto

```

```

lemma get-rev-level-map-convert:
  get-rev-level x n (map convert M) = get-rev-level x n M
  by (induction M arbitrary; n rule: marked-lit-list-induct) auto

```

```

lemma get-level-map-convert[simp]:
  get-level x (map convert M) = get-level x M
  using get-rev-level-map-convert[of x 0 rev M] by (simp add: rev-map)

```

```

lemma get-maximum-level-map-convert[simp]:
  get-maximum-level D (map convert M) = get-maximum-level D M
  by (induction D)

```

(*auto simp add: get-maximum-level-plus*)

**lemma** *get-all-levels-of-marked-map-convert*[simp]:  
*get-all-levels-of-marked* (map convert *M*) = (*get-all-levels-of-marked* *M*)  
**by** (*induction M rule: marked-lit-list-induct*) *auto*

Conversion function

**fun** *toS* :: *cdcl<sub>W</sub>-state-inv-st*  $\Rightarrow$  *nat 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** *cdcl<sub>W</sub>-state-inv* = {*S*::*cdcl<sub>W</sub>-state-inv-st*. *cdcl<sub>W</sub>-all-struct-inv* (*toS S*)}  
**morphisms** *rough-state-of state-of*  
**proof**  
**show** ( $\square, \square, \square, 0, C\text{-True}$ )  $\in$  {*S*. *cdcl<sub>W</sub>-all-struct-inv* (*toS S*)}  
**by** (*auto simp add: cdcl<sub>W</sub>-all-struct-inv-def*)  
**qed**

**instantiation** *cdcl<sub>W</sub>-state-inv* :: *equal*

**begin**

**definition** *equal-cdcl<sub>W</sub>-state-inv* :: *cdcl<sub>W</sub>-state-inv*  $\Rightarrow$  *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**

**by** *standard* (*simp add: rough-state-of-inject equal-cdcl<sub>W</sub>-state-inv-def*)  
**end**

**lemma** *lits-of-map-convert*[simp]: *lits-of* (map convert *M*) = *lits-of M*  
**by** (*induction M rule: marked-lit-list-induct*) *simp-all*

**lemma** *undefined-lit-map-convert*[iff]:  
*undefined-lit* (map convert *M*) *L*  $\longleftrightarrow$  *undefined-lit M L*  
**by** (*auto simp add: Marked-Propagated-in-iff-in-lits-of*)

**lemma** *true-annot-map-convert*[simp]: map convert *M*  $\models_a N \longleftrightarrow M \models_a N$   
**by** (*induction M rule: marked-lit-list-induct*) (*simp-all add: true-annot-def*)

**lemma** *true-annots-map-convert*[simp]: map convert *M*  $\models_{as} N \longleftrightarrow M \models_{as} N$   
**unfolding** *true-annots-def* **by** *auto*

**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, C-True)*) (*toS (Propagated L C # M, N, U, k, C-True)*)  
**using** *assms*  
**by** (*auto dest!: find-first-unit-clause-some simp add: propagate.simps*  
*intro!: exI[of - mset C - {#L#}]*)

### 18.3.2 Propagate

**definition** *do-propagate-step* **where**

*do-propagate-step S* =  
(*case S of*  
(*M, N, U, k, C-True*)  $\Rightarrow$   
(*case find-first-unit-clause (N @ U) M of*

$\text{Some } (L, C) \Rightarrow (\text{Propagated } L \ C \ \# \ M, N, U, k, C\text{-True})$   
 $\mid \text{None} \Rightarrow (M, N, U, k, C\text{-True})$   
 $\mid S \Rightarrow S$

**lemma** *do-propagate-step*:

$\text{do-propagate-step } S \neq S \Rightarrow \text{propagate } (\text{toS } S) (\text{toS } (\text{do-propagate-step } S))$   
**apply** (cases *S*, cases *conflicting S*)  
**using** *find-first-unit-clause-some-is-propagate*[of clauses *S* learned-clss *S* trail *S* - -  
backtrack-lvl *S*]  
**by** (auto simp add: *do-propagate-step-def split: option.splits*)

**lemma** *do-propagate-step-conflicting-clause*[simp]:

$\text{conflicting } S \neq C\text{-True} \Rightarrow \text{do-propagate-step } S = S$   
**unfolding** *do-propagate-step-def* **by** (cases *S*, cases *conflicting S*) auto

**lemma** *do-propagate-step-no-step*:

**assumes** *dist*:  $\forall c \in \text{set } (\text{clauses } S \ @ \ \text{learned-clss } S). \text{ distinct } c$  **and**  
*prop-step*:  $\text{do-propagate-step } S = S$   
**shows** *no-step propagate* (toS *S*)

**proof** (standard, standard)

**fix** *T*  
**assume** *propagate* (toS *S*) *T*  
**then obtain** *M N U k C L* **where**  
 $\text{toSS: toS } S = (M, N, U, k, C\text{-True})$  **and**  
 $T: T = (\text{Propagated } L \ (C + \{\#L\}) \ \# \ M, N, U, k, C\text{-True})$  **and**  
 $MC: M \models_{\text{as}} C\text{Not } C$  **and**  
*undef*: *undefined-lit M L* **and**  
 $CL: C + \{\#L\} \in \# \ N + U$   
**apply** - **by** (cases toS *S*) auto  
**let** *?M* = trail *S*  
**let** *?N* = clauses *S*  
**let** *?U* = learned-clss *S*  
**let** *?k* = backtrack-lvl *S*  
**let** *?D* = *C-True*  
**have** *S*:  $S = (?M, ?N, ?U, ?k, ?D)$   
**using** *toSS* **by** (cases *S*, cases *conflicting S*) simp-all  
**have** *S*:  $\text{toS } S = \text{toS } (?M, ?N, ?U, ?k, ?D)$   
**unfolding** *S[symmetric]* **by** simp

**have**

$M: M = \text{map convert } ?M$  **and**  
 $N: N = \text{mset } (\text{map mset } ?N)$  **and**  
 $U: U = \text{mset } (\text{map mset } ?U)$   
**using** *toSS[unfolded S]* **by** auto

**obtain** *D* **where**

$DCL: \text{mset } D = C + \{\#L\}$  **and**  
 $D: D \in \text{set } (?N \ @ \ ?U)$   
**using** *CL* **unfolding** *N U* **by** auto

**obtain** *C' L'* **where**

$\text{setD: set } D = \text{set } (L' \ \# \ C')$  **and**  
 $C': \text{mset } C' = C$  **and**  
 $L: L = L'$   
**using** *DCL* **by** (metis *ex-mset mset.simps(2) mset-eq-setD*)

**have** *find-first-unit-clause* (?N @ ?U) ?M  $\neq$  None

```

apply (rule dist find-first-unit-clause-none[of  $D \text{ ?}N @ \text{ ?}U \text{ ?}M L$ ,  $OF - D$ ])
  using  $D$  assms(1) apply auto[1]
  using  $MC$  setD DCL M MC unfolding  $C'$ [symmetric] apply auto[1]
  using  $M$  undef apply auto[1]
  unfolding setD L by auto
then show False using prop-step S unfolding do-propagate-step-def by (cases S) auto
qed

```

**Conflict** **fun** *find-conflict* **where**

*find-conflict M []* = *None* |

*find-conflict M (N # Ns)* = (if ( $\forall c \in \text{set } N. -c \in \text{lits-of } M$ ) then *Some N* else *find-conflict M Ns*)

**lemma** *find-conflict-Some*:

*find-conflict M Ns* = *Some N*  $\implies N \in \text{set } Ns \wedge M \models_{as} CNot \text{ (mset } N)$

**by** (*induction Ns* rule: *find-conflict.induct*)

(*auto split: split-if-asm*)

**lemma** *find-conflict-None*:

*find-conflict M Ns* = *None*  $\longleftrightarrow (\forall N \in \text{set } Ns. \neg M \models_{as} CNot \text{ (mset } N))$

**by** (*induction Ns*) *auto*

**lemma** *find-conflict-None-no-conf*:

*find-conflict M (N@U)* = *None*  $\longleftrightarrow \text{no-step conflict (toS (M, N, U, k, C-True))}$

**by** (*auto simp add: find-conflict-None conflict.simps*)

**definition** *do-conflict-step* **where**

*do-conflict-step S* =

(*case S* of

( $M, N, U, k, C\text{-True}$ )  $\Rightarrow$

(*case find-conflict M (N @ U)* of

*Some a*  $\Rightarrow (M, N, U, k, C\text{-Clause } a)$

| *None*  $\Rightarrow (M, N, U, k, C\text{-True}))$

|  $S \Rightarrow S$ )

**lemma** *do-conflict-step*:

*do-conflict-step S*  $\neq S \implies \text{conflict (toS } S) \text{ (toS (do-conflict-step } S))}$

**apply** (*cases S, cases conflicting S*)

**unfolding** *conflict.simps do-conflict-step-def*

**by** (*auto dest!: find-conflict-Some split: option.splits*)

**lemma** *do-conflict-step-no-step*:

*do-conflict-step S* = *S*  $\implies \text{no-step conflict (toS } S)$

**apply** (*cases S, cases conflicting S*)

**unfolding** *do-conflict-step-def*

**using** *find-conflict-None-no-conf*[of *trail S clauses S learned-clss S*  
*backtrack-lvl S*]

**by** (*auto split: option.splits*)

**lemma** *do-conflict-step-conflicting-clause[simp]*:

*conflicting S*  $\neq C\text{-True} \implies \text{do-conflict-step } S = S$

**unfolding** *do-conflict-step-def* **by** (*cases S, cases conflicting S*) *auto*

**lemma** *do-conflict-step-conflicting[dest]*:

*do-conflict-step S*  $\neq S \implies \text{conflicting (do-conflict-step } S) \neq C\text{-True}$

**unfolding** *do-conflict-step-def* **by** (*cases S, cases conflicting S*) (*auto split: option.splits*)

**definition** *do-cp-step* where

*do-cp-step*  $S =$

(*do-propagate-step*  $\circ$  *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* (*toS*  $S$ ) (*toS* (*do-cp-step*  $S$ ))

**proof** –

show ?thesis

**proof** (cases *do-conflict-step*  $S \neq S$ )

case *True*

then show ?thesis

by (auto simp add: *do-conflict-step* *do-conflict-step-conflicting* *do-cp-step-def*)

next

case *False*

then have *confl*[*simp*]: *do-conflict-step*  $S = S$  by *simp*

show ?thesis

**proof** (cases *do-propagate-step*  $S = S$ )

case *True*

then show ?thesis

using  $H$  by (simp add: *do-cp-step-def*)

next

case *False*

let ? $S =$  *toS*  $S$

let ? $T =$  *toS* (*do-propagate-step*  $S$ )

let ? $U =$  *toS* (*do-conflict-step* (*do-propagate-step*  $S$ ))

have *propa*: *propagate* (*toS*  $S$ ) ? $T$  using *False* *do-propagate-step* by blast

moreover have *ns*: *no-step conflict* (*toS*  $S$ ) using *confl* *do-conflict-step-no-step* by blast

ultimately show ?thesis

using *cdcl<sub>W</sub>-cp.intros*(2)[of ? $S$  ? $T$ ] *confl* **unfolding** *do-cp-step-def* by auto

qed

qed

qed

**lemma** *do-cp-step-eq-no-prop-no-confl*:

*do-cp-step*  $S = S \implies$  *do-conflict-step*  $S = S \wedge$  *do-propagate-step*  $S = S$

by (cases  $S$ , cases *conflicting*  $S$ )

(auto simp add: *do-conflict-step-def* *do-propagate-step-def* *do-cp-step-def* *split*: *option.splits*)

**lemma** *no-cdcl<sub>W</sub>-cp-iff-no-propagate-no-conflict*:

*no-step* *cdcl<sub>W</sub>-cp*  $S \longleftrightarrow$  *no-step* *propagate*  $S \wedge$  *no-step* *conflict*  $S$

by (auto simp: *cdcl<sub>W</sub>-cp.simps*)

**lemma** *do-cp-step-eq-no-step*:

assumes  $H$ : *do-cp-step*  $S = S$  and  $\forall c \in$  set (clauses  $S$  @ *learned-cls*  $S$ ). *distinct*  $c$

shows *no-step* *cdcl<sub>W</sub>-cp* (*toS*  $S$ )

**unfolding** *no-cdcl<sub>W</sub>-cp-iff-no-propagate-no-conflict*

using *assms* **apply** (cases  $S$ , cases *conflicting*  $S$ )

using *do-propagate-step-no-step*[of  $S$ ]

by (auto dest!: *do-cp-step-eq-no-prop-no-confl*[*simplified*] *do-conflict-step-no-step* *split*: *option.splits*)

**lemma** *cdcl<sub>W</sub>-cp-cdcl<sub>W</sub>-st*: *cdcl<sub>W</sub>-cp*  $S S' \implies$  *cdcl<sub>W</sub>\*\**  $S S'$

by (simp add: *cdcl<sub>W</sub>-cp-tranclp-cdcl<sub>W</sub>* *tranclp-into-rtranclp*)

**lemma** *cdcl<sub>W</sub>-cp-wf-all-inv*: wf {(S', S::'v::linorder *cdcl<sub>W</sub>-state*). *cdcl<sub>W</sub>-all-struct-inv* S ∧ *cdcl<sub>W</sub>-cp* S S'}  
 (is wf ?R)  
**proof** (rule wf-bounded-measure[of - λS. card (atms-of-mu (clauses S))+1  
 λS. length (trail S) + (if conflicting S = C-True then 0 else 1)], goal-cases)  
 case (1 S S')  
 then have *cdcl<sub>W</sub>-all-struct-inv* S and *cdcl<sub>W</sub>-cp* S S' by auto  
 moreover then have *cdcl<sub>W</sub>-all-struct-inv* S'  
 using rtrancp-cdcl<sub>W</sub>-all-struct-inv-inv *cdcl<sub>W</sub>-cp-cdcl<sub>W</sub>-st* by blast  
 ultimately show ?case  
 by (auto simp add: *cdcl<sub>W</sub>-cp.simps* elim!: conflictE propagateE  
 dest: length-model-le-vars-all-inv)  
**qed**

**lemma** *cdcl<sub>W</sub>-all-struct-inv-rough-state[simp]*: *cdcl<sub>W</sub>-all-struct-inv* (toS (rough-state-of S))  
 using rough-state-of by auto

**lemma** [simp]: *cdcl<sub>W</sub>-all-struct-inv* (toS S) ⇒ rough-state-of (state-of S) = S  
 by (simp add: state-of-inverse)

**lemma** *rough-state-of-state-of-do-cp-step[simp]*:  
 rough-state-of (state-of (do-cp-step (rough-state-of S))) = do-cp-step (rough-state-of S)  
**proof** –  
 have *cdcl<sub>W</sub>-all-struct-inv* (toS (do-cp-step (rough-state-of S)))  
 apply (cases do-cp-step (rough-state-of S) = (rough-state-of S))  
 apply simp  
 using cp-step-is-cdcl<sub>W</sub>-cp[of rough-state-of S]  
 cdcl<sub>W</sub>-all-struct-inv-rough-state[of S] *cdcl<sub>W</sub>-cp-cdcl<sub>W</sub>-st* rtrancp-cdcl<sub>W</sub>-all-struct-inv-inv by blast  
 then show ?thesis by auto  
**qed**

**Skip fun** *do-skip-step* :: *cdcl<sub>W</sub>-state-inv-st* ⇒ *cdcl<sub>W</sub>-state-inv-st* **where**  
*do-skip-step* (Propagated L C # Ls, N, U, k, C-Clause D) =  
 (if ¬L ∈ set D ∧ D ≠ []  
 then (Ls, N, U, k, C-Clause D)  
 else (Propagated L C # Ls, N, U, k, C-Clause D)) |  
*do-skip-step* S = S

**lemma** *do-skip-step*:  
*do-skip-step* S ≠ S ⇒ skip (toS S) (toS (do-skip-step S))  
 apply (induction S rule: *do-skip-step.induct*)  
 by (auto simp add: skip.simps)

**lemma** *do-skip-step-no*:  
*do-skip-step* S = S ⇒ no-step skip (toS S)  
 by (induction S rule: *do-skip-step.induct*)  
 (auto simp add: other split: split-if-asm)

**lemma** *do-skip-step-trail-is-C-True[iff]*:  
*do-skip-step* S = (a, b, c, d, C-True) ⇔ S = (a, b, c, d, C-True)  
 by (cases S rule: *do-skip-step.cases*) auto

**Resolve fun** *maximum-level-code*:: 'a literal list ⇒ ('a, nat, 'a literal list) marked-lit list ⇒ nat **where**  
*maximum-level-code* [] = 0 |

$\text{maximum-level-code } (L \# Ls) \ M = \max (\text{get-level } L \ M) (\text{maximum-level-code } Ls \ M)$

**lemma** *maximum-level-code-eq-get-maximum-level*[code, simp]:  
 $\text{maximum-level-code } D \ M = \text{get-maximum-level } (\text{mset } D) \ M$   
**by** (induction D) (auto simp add: get-maximum-level-plus)

**fun** *do-resolve-step* ::  $\text{cdcl}_W\text{-state-inv-st} \Rightarrow \text{cdcl}_W\text{-state-inv-st}$  **where**  
*do-resolve-step* (Propagated L C # Ls, N, U, k, C-Clause D) =  
 (if  $-L \in \text{set } D \wedge (\text{maximum-level-code } (\text{remove1 } (-L) \ D) (\text{Propagated } L \ C \ \# \ Ls) = k \vee k = 0)$   
 then (Ls, N, U, k, C-Clause (remdups (remove1 L C @ remove1 (-L) D)))  
 else (Propagated L C # Ls, N, U, k, C-Clause D)) |  
*do-resolve-step* S = S

**lemma** *distinct-mset-remdups-union-mset*:  
**assumes** *distinct-mset A and distinct-mset B*  
**shows**  $A \ \# \cup \ B = \text{remdups-mset } (A + B)$   
**using** *assms unfolding remdups-mset-def apply* (auto simp: multiset-eq-iff max-def)  
**apply** (metis Un-iff count-mset-set(1) count-mset-set(3) distinct-mset-set-mset-ident  
 finite-UnI finite-set-mset mem-set-mset-iff not-le)  
**by** (simp add: distinct-mset-def)

**lemma** *do-resolve-step*:  
 $\text{cdcl}_W\text{-all-struct-inv } (\text{toS } S) \Longrightarrow \text{do-resolve-step } S \neq S$   
 $\Longrightarrow \text{resolve } (\text{toS } S) (\text{toS } (\text{do-resolve-step } S))$

**proof** (induction S rule: do-resolve-step.induct)

**case** (1 L C M N U k D)

**moreover**

{ **assume** [simp]:  $k = 0$   
**have**  $\text{get-all-levels-of-marked } (\text{Propagated } L \ C \ \# \ M) = []$   
**using** 1(1) **unfolding**  $\text{cdcl}_W\text{-all-struct-inv-def } \text{cdcl}_W\text{-M-level-inv-def}$  **by** simp  
**then have**  $H: \bigwedge L'. \text{get-level } L' (\text{Propagated } L \ C \ \# \ M) = 0$   
**by** (metis (no-types, hide-lams) Un-insert-left empty-iff get-all-levels-of-marked.simps(3)  
 get-level-in-levels-of-marked insert-iff list.set(1) sup-bot.left-neutral)  
**}** **note**  $H = \text{this}$

**ultimately have**

$- L \in \text{set } D$  **and**

$M: \text{maximum-level-code } (\text{remove1 } (-L) \ D) (\text{Propagated } L \ C \ \# \ M) = k$

**by** (cases mset D - {#- L#} = {#},

auto dest!: get-maximum-level-exists-lit-of-max-level[of - Propagated L C # M]

split: split-if-asm simp add: H)+

**have** *every-mark-is-a-conflict* (toS (Propagated L C # M, N, U, k, C-Clause D))

**using** 1(1) **unfolding**  $\text{cdcl}_W\text{-all-struct-inv-def } \text{cdcl}_W\text{-conflicting-def}$  **by** fast

**then have**  $L \in \text{set } C$  **by** fastforce

**then obtain** C' **where** C:  $\text{mset } C = C' + \{\#L\# \}$

**by** (metis add.commute in-multiset-in-set insert-DiffM)

**obtain** D' **where** D:  $\text{mset } D = D' + \{\#-L\# \}$

**using**  $\langle - L \in \text{set } D \rangle$  **by** (metis add.commute in-multiset-in-set insert-DiffM)

**have** D'L:  $D' + \{\#-L\# \} - \{\#-L\# \} = D'$  **by** (auto simp add: multiset-eq-iff)

**have** CL:  $\text{mset } C - \{\#L\# \} + \{\#L\# \} = \text{mset } C$  **using**  $\langle L \in \text{set } C \rangle$  **by** (auto simp add: multiset-eq-iff)

**have**

*resolve*

(map convert (Propagated L C # M), mset '# mset N, mset '# mset U, k, C-Clause (mset D))

(map convert M, mset '# mset N, mset '# mset U, k,



```

      C-Clause (((mset D - {#-L#}) #∪ (mset C - {#L#}))))
unfolding resolve.simps
  apply (simp add: C D)
using M[simplified] unfolding maximum-level-code-eq-get-maximum-level C[symmetric] CL
by (metis D D'L convert.simps(1) get-maximum-level-map-convert list.simps(9))
moreover have
  (map convert (Propagated L C # M), mset '# mset N, mset '# mset U, k, C-Clause (mset D))
  = toS (Propagated L C # M, N, U, k, C-Clause D)
by auto
moreover
  have distinct-mset (mset C) and distinct-mset (mset D)
    using ⟨cdclW-all-struct-inv (toS (Propagated L C # M, N, U, k, C-Clause D))⟩
    unfolding cdclW-all-struct-inv-def distinct-cdclW-state-def
    by auto
  then have (mset C - {#L#}) #∪ (mset D - {#- L#}) =
    remdups-mset (mset C - {#L#} + (mset D - {#- L#}))
    apply -
    apply (rule distinct-mset-rempdups-union-mset)
    by auto
  then have (map convert M, mset '# mset N, mset '# mset U, k,
    C-Clause (((mset D - {#- L#}) #∪ (mset C - {#L#}))))
  = toS (do-resolve-step (Propagated L C # M, N, U, k, C-Clause D))
    using ⟨- L ∈ set D⟩ M by (auto simp:ac-simps )
ultimately show ?case
by simp
qed auto

```

**lemma** do-resolve-step-no:

```

do-resolve-step S = S ⟹ no-step resolve (toS S)
apply (cases S; cases hd (trail S); cases conflicting S)
by (auto
  elim!: resolveE split: split-if-asm
  dest!: union-single-eq-member
  simp del: in-multiset-in-set get-maximum-level-map-convert
  simp add: in-multiset-in-set[symmetric] get-maximum-level-map-convert[symmetric])

```

**lemma** rough-state-of-state-of-resolve[simp]:

```

cdclW-all-struct-inv (toS S) ⟹ rough-state-of (state-of (do-resolve-step S)) = do-resolve-step S
apply (rule state-of-inverse)
by (smt CollectI bj cdclW-all-struct-inv-inv do-resolve-step other resolve)

```

**lemma** do-resolve-step-trail-is-C-True[iff]:

```

do-resolve-step S = (a, b, c, d, C-True) ⟷ S = (a, b, c, d, C-True)
by (cases S rule: do-resolve-step.cases)
  auto

```

**Backjumping** **fun** find-level-decomp **where**

```

find-level-decomp M [] D k = None |
find-level-decomp M (L # Ls) D k =
  (case (get-level L M, maximum-level-code (D @ Ls) M) of
    (i, j) ⇒ if i = k ∧ 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 ∈ set Ls ∧ get-maximum-level (mset (remove1 L (Ls @ D))) M = j ∧ get-level L M = k
using assms
apply (induction Ls arbitrary: D)
apply simp
apply (auto split: split-if-asm simp add: ac-simps)
apply (smt ab-semigroup-add-class.add-ac(1) add.commute diff-union-swap mset.simps(2))
apply (smt add.commute add.left-commute diff-union-cancelL mset.simps(2))
apply (smt add.commute add.left-commute diff-union-swap mset.simps(2))
done

```

**lemma** find-level-decomp-none:

```

assumes find-level-decomp M Ls E k = None and mset (L#D) = mset (Ls @ E)
shows ¬(L ∈ set Ls ∧ get-maximum-level (mset D) M < k ∧ k = get-level L M)
using assms

```

**proof** (induction Ls arbitrary: E L D)

**case** Nil

**then show** ?case **by** simp

**next**

**case** (Cons L' Ls) **note** IH = this(1) **and** find-none = this(2) **and** LD = this(3)

```

have mset D + {#L'#} = mset E + (mset Ls + {#L'#}) ⇒ mset D = mset E + mset Ls
by (metis add-right-imp-eq union-assoc)

```

**then show** ?case

**using** find-none IH[of L' # E L D] LD **by** (auto simp add: ac-simps split: split-if-asm)

**qed**

**fun** bt-cut **where**

bt-cut i (Propagated - - # Ls) = bt-cut i Ls |

bt-cut i (Marked K k # Ls) = (if k = Suc i then Some (Marked K k # Ls) else bt-cut i Ls) |

bt-cut i [] = None

**lemma** bt-cut-some-decomp:

```

bt-cut i M = Some M' ⇒ ∃ K M2 M1. M = M2 @ M' ∧ M' = Marked K (i+1) # M1
by (induction i M rule: bt-cut.induct) (auto split: split-if-asm)

```

**lemma** bt-cut-not-none: M = M2 @ Marked K (Suc i) # M' ⇒ bt-cut i M ≠ None

**by** (induction M2 arbitrary: M rule: marked-lit-list-induct) auto

**lemma** get-all-marked-decomposition-ex:

∃ N. (Marked K (Suc i) # M', N) ∈ set (get-all-marked-decomposition (M2@Marked K (Suc i) # M'))

**apply** (induction M2 rule: marked-lit-list-induct)

**apply** auto[2]

**by** (case-tac get-all-marked-decomposition (xs @ Marked K (Suc i) # M')) auto

**lemma** bt-cut-in-get-all-marked-decomposition:

bt-cut i M = Some M' ⇒ ∃ M2. (M', M2) ∈ set (get-all-marked-decomposition M)

**by** (auto dest!: bt-cut-some-decomp simp add: get-all-marked-decomposition-ex)

**fun** do-backtrack-step **where**

do-backtrack-step (M, N, U, k, C-Clause D) =

(case find-level-decomp M D [] k of

None ⇒ (M, N, U, k, C-Clause D)

| Some (L, j) ⇒

(case bt-cut j M of

```

    Some (Marked - - # Ls) ⇒ (Propagated L D # Ls, N, D # U, j, C-True)
  | - ⇒ (M, N, U, k, C-Clause D))
) |
do-backtrack-step S = S

```

**lemma** *get-all-marked-decomposition-map-convert*:  
 (get-all-marked-decomposition (map convert M)) =  
 map (λ(a, b). (map convert a, map convert b)) (get-all-marked-decomposition M)  
**apply** (induction M rule: marked-lit-list-induct)  
**apply** simp  
**by** (case-tac get-all-marked-decomposition xs, auto)+

**lemma** *do-backtrack-step*:  
**assumes** db: do-backtrack-step S ≠ S  
**and** inv: cdcl<sub>W</sub>-all-struct-inv (toS S)  
**shows** backtrack (toS S) (toS (do-backtrack-step S))  
**proof** (cases S, cases conflicting S, goal-cases)  
 case (1 M N U k E)  
 then show ?case **using** db **by** auto  
**next**  
 case (2 M N U k E C) **note** S = this(1) **and** confl = this(2)  
 have E: E = C-Clause C **using** S confl **by** auto  
  
**obtain** L j **where** fd: find-level-decomp M C [] k = Some (L, j)  
**using** db **unfolding** S E **by** (cases C) (auto split: split-if-asm option.splits)  
**have** L ∈ set C **and** get-maximum-level (mset (remove1 L C)) M = j **and**  
 levL: get-level L M = k  
**using** find-level-decomp-some[OF fd] **by** auto  
**obtain** C' **where** C: mset C = mset C' + {#L#}  
**using** ⟨L ∈ set C⟩ **by** (metis add.commute ex-mset in-multiset-in-set insert-DiffM)  
**obtain** M<sub>2</sub> **where** M<sub>2</sub>: bt-cut j M = Some M<sub>2</sub>  
**using** db fd **unfolding** S E **by** (auto split: option.splits)  
**obtain** M1 K **where** M1: M<sub>2</sub> = Marked K (Suc j) # M1  
**using** bt-cut-some-decomp[OF M<sub>2</sub>] **by** (cases M<sub>2</sub>) auto  
**obtain** c **where** c: M = c @ Marked K (Suc j) # M1  
**using** bt-cut-in-get-all-marked-decomposition[OF M<sub>2</sub>]  
**unfolding** M1 **by** fastforce  
**have** get-all-levels-of-marked (map convert M) = rev [1..<sub>Suc</sub> k]  
**using** inv **unfolding** cdcl<sub>W</sub>-all-struct-inv-def cdcl<sub>W</sub>-M-level-inv-def S **by** auto  
**from** arg-cong[OF this, of λa. Suc j ∈ set a] **have** j ≤ k **unfolding** c **by** auto  
**have** max-l-j: maximum-level-code C' M = j  
**using** db fd M<sub>2</sub> C **unfolding** S E **by** (auto  
 split: option.splits list.splits marked-lit.splits  
 dest!: find-level-decomp-some)[1]  
**have** get-maximum-level (mset C) M ≥ k  
**using** ⟨L ∈ set C⟩ get-maximum-level-ge-get-level levL **by** blast  
**moreover** **have** get-maximum-level (mset C) M ≤ k  
**using** get-maximum-level-exists-lit-of-max-level[of mset C M] inv  
 cdcl<sub>W</sub>-M-level-inv-get-level-le-backtrack-lvl[of toS S]  
**unfolding** C cdcl<sub>W</sub>-all-struct-inv-def S  
**by** auto metis+  
**ultimately** **have** get-maximum-level (mset C) M = k **by** auto  
  
**obtain** M2 **where** M2: (M<sub>2</sub>, M2) ∈ set (get-all-marked-decomposition M)  
**using** bt-cut-in-get-all-marked-decomposition[OF M<sub>2</sub>] **by** metis

```

have H: (cdclW.reduce-trail-to (map convert M1)
  (add-learned-cls (mset C' + {#L#})
    (map convert M, mset (map mset N), mset (map mset U), j, C-True))) =
  (map convert M1, mset (map mset N), {#mset C' + {#L#}#} + mset (map mset U), j, C-True)
  apply (subst state-conv[of cdclW.reduce-trail-to - -])
using M2 unfolding M1 by auto
have
  backtrack
    (map convert M, mset '# mset N, mset '# mset U, k, C-Clause (mset C))
    (Propagated L (mset C) # map convert M1, mset '# mset N, mset '# mset U + {#mset C#},
      j,
        C-True)
  apply (rule backtrack-rule)
    unfolding C apply simp
    using Set.imageI[of (M2, M2) set (get-all-marked-decomposition M)
      (λ(a, b). (map convert a, map convert b))] M2
    apply (auto simp: get-all-marked-decomposition-map-convert M1)[1]
    using max-l-j levL ⟨j ≤ k⟩ apply (simp add: get-maximum-level-plus)
    using C ⟨get-maximum-level (mset C) M = k⟩ levL apply auto[1]
    using max-l-j apply simp
  apply (cases cdclW.reduce-trail-to (map convert M1)
    (add-learned-cls (mset C' + {#L#})
      (map convert M, mset (map mset N), mset (map mset U), j, C-True)))
    using M2 M1 H by (auto simp: ac-simps)
  then show ?case
    using M2 fd unfolding S E M1 by auto
  obtain M2 where (M2, M2) ∈ set (get-all-marked-decomposition M)
    using bt-cut-in-get-all-marked-decomposition[OF M2] by metis
qed

```

**lemma** *do-backtrack-step-no*:

```

assumes db: do-backtrack-step S = S
and inv: cdclW-all-struct-inv (toS S)
shows no-step backtrack (toS S)
proof (rule ccontr, cases S, cases conflicting S, goal-cases)
  case 1
  then show ?case using db by (auto split: option.splits)
next
  case (2 M N U k E C) note bt = this(1) and S = this(2) and confl = this(3)
  obtain D L K b z M1 j where
    levL: get-level L M = get-maximum-level (D + {#L#}) M and
    k: k = get-maximum-level (D + {#L#}) M and
    j: j = get-maximum-level D M and
    CE: convertC E = C-Clause (D + {#L#}) and
    decomp: (z # M1, b) ∈ set (get-all-marked-decomposition M) and
    z: Marked K (Suc j) = convert z using bt unfolding S
    by (auto split: option.splits elim!: backtrackE
      simp: get-all-marked-decomposition-map-convert)
  have z: z = Marked K (Suc j) using z by (cases z) auto
  obtain c where c: M = c @ b @ Marked K (Suc j) # M1
    using decomp unfolding z by blast
  have get-all-levels-of-marked (map convert M) = rev [1..Suc k]
    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def S by auto
  from arg-cong[OF this, of λa. Suc j ∈ set a] have k > j unfolding c by auto
  obtain C D' where

```

```

E: E = C-Clause C and
C: mset C = mset (L # D')
using CE apply (cases E)
  apply simp
  by (metis conflicting-clause.inject convertC.simps(1) ex-mset mset.simps(2))
have D'D: mset D' = D
  using C CE E by auto
have find-level-decomp M C [] k ≠ None
  apply rule
  apply (drule find-level-decomp-none[of - - - L D'])
  using C ⟨k > j⟩ mset-eq-setD unfolding k[symmetric] D'D j[symmetric] levL by fastforce+
then obtain L' j' where fd-some: find-level-decomp M C [] k = Some (L', j')
  by (cases find-level-decomp M C [] k) auto
have L': L' = L
  proof (rule ccontr)
    assume ¬ ?thesis
    then have L' ∈# D
      by (metis C D'D fd-some find-level-decomp-some in-multiset-in-set insert-iff list.simps(15))
    then have get-level L' M ≤ get-maximum-level D M
      using get-maximum-level-ge-get-level by blast
    then show False using ⟨k > j⟩ j find-level-decomp-some[OF fd-some] by auto
  qed
then have j': j' = j using find-level-decomp-some[OF fd-some] j C D'D by auto

have btc-none: bt-cut j M ≠ None
  apply (rule bt-cut-not-none[of M - @ -])
  using c by simp
show ?case using db unfolding S E
  by (auto split: option.splits list.splits marked-lit.splits
    simp add: fd-some L' j' btc-none
    dest: bt-cut-some-decomp)
qed

lemma rough-state-of-state-of-backtrack[simp]:
  assumes inv: cdclW-all-struct-inv (toS S)
  shows rough-state-of (state-of (do-backtrack-step S)) = do-backtrack-step S
proof (rule state-of-inverse)
  have f2: backtrack (toS S) (toS (do-backtrack-step S)) ∨ do-backtrack-step S = S
    using do-backtrack-step inv by blast
  have ∧p. ¬ cdclW-o (toS S) p ∨ cdclW-all-struct-inv p
    using inv cdclW-all-struct-inv-inv other by blast
  then have do-backtrack-step S = S ∨ cdclW-all-struct-inv (toS (do-backtrack-step S))
    using f2 by blast
  then show do-backtrack-step S ∈ {S. cdclW-all-struct-inv (toS S)}
    using inv by fastforce
qed

Decide fun do-decide-step where
do-decide-step (M, N, U, k, C-True) =
  (case find-first-unused-var N (lits-of M) of
    None ⇒ (M, N, U, k, C-True)
  | Some L ⇒ (Marked L (Suc k) # M, N, U, k+1, C-True)) |
do-decide-step S = S

lemma do-decide-step:

```

```

do-decide-step  $S \neq S \implies \text{decide } (\text{toS } S) (\text{toS } (\text{do-decide-step } S))$ 
apply (cases  $S$ , cases conflicting  $S$ )
defer
apply (auto split: option.splits simp add: decide.simps Marked-Propagated-in-iff-in-lits-of
  dest: find-first-unused-var-undefined find-first-unused-var-Some
  intro: atms-of-atms-of-m-mono)[1]
proof –
  fix a b c d e
  {
    fix a :: (nat, nat, nat literal list) marked-lit list and
      b :: nat literal list list and c :: nat literal list list and
      d :: nat and x2 :: nat literal and m :: nat literal list
    assume a1:  $m \in \text{set } b$ 
    assume x2  $\in \text{set } m$ 
    then have f2:  $\text{atm-of } x2 \in \text{atms-of } (\text{mset } m)$ 
      by simp
    have  $\bigwedge f. (f \text{ m} :: \text{nat literal multiset}) \in f \text{ ' set } b$ 
      using a1 by blast
    then have  $\bigwedge f. (\text{atms-of } (f \text{ m}) :: \text{nat set}) \subseteq \text{atms-of-m } (f \text{ ' set } b)$ 
      using atms-of-atms-of-m-mono by blast
    then have  $\bigwedge n f. (n :: \text{nat}) \in \text{atms-of-m } (f \text{ ' set } b) \vee n \notin \text{atms-of } (f \text{ m})$ 
      by (meson contra-subsetD)
    then have  $\text{atm-of } x2 \in \text{atms-of-m } (\text{mset ' set } b)$ 
      using f2 by blast
  } note H = this
assume do-decide-step  $S \neq S$  and
   $S = (a, b, c, d, e)$  and
  conflicting  $S = \text{C-True}$ 
then show  $\text{decide } (\text{toS } S) (\text{toS } (\text{do-decide-step } S))$ 

  apply (auto split: option.splits simp add: decide.simps Marked-Propagated-in-iff-in-lits-of
    dest!: find-first-unused-var-Some dest: H)
  by (meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI)+
qed

```

**lemma** do-decide-step-no:

```

do-decide-step  $S = S \implies \text{no-step decide } (\text{toS } S)$ 
apply (cases  $S$ , cases conflicting  $S$ )
apply (auto
  simp add: atms-of-m-mset-unfold atm-of-eq-atm-of Marked-Propagated-in-iff-in-lits-of
  split: option.splits
  elim!: decideE)
apply (meson atm-of-in-atm-of-set-in-uminus image-subset-iff)
apply (meson atm-of-in-atm-of-set-in-uminus image-subset-iff)
done

```

**lemma** rough-state-of-state-of-do-decide-step[simp]:

```

cdclW-all-struct-inv (toS  $S$ )  $\implies \text{rough-state-of } (\text{state-of } (\text{do-decide-step } S)) = \text{do-decide-step } S$ 
apply (subst state-of-inverse)
apply (smt cdclW-all-struct-inv-inv decide do-decide-step mem-Collect-eq other)
apply simp
done

```

```

lemma rough-state-of-state-of-do-skip-step[simp]:
   $cdcl_W\text{-all-struct-inv } (toS\ S) \implies \text{rough-state-of } (state\text{-of } (do\text{-skip-step } S)) = do\text{-skip-step } S$ 
apply (subst state-of-inverse)
apply (smt cdcl_W-all-struct-inv-inv skip do-skip-step mem-Collect-eq other bj)
apply simp
done

```

### 18.3.3 Code generation

**Type definition** There are two invariants: one while applying conflict and propagate and one for the other rules

```

declare rough-state-of-inverse[simp add]

```

**definition** *Con* **where**

```

Con xs = state-of (if cdcl_W-all-struct-inv (toS (fst xs, snd xs)) then xs
else ([], [], [], 0, C-True))

```

**lemma** [*code abstype*]:

```

Con (rough-state-of S) = S

```

```

using rough-state-of[of S] unfolding Con-def by (simp add: rough-state-of-inverse)

```

**definition** *do-cp-step'* **where**

```

do-cp-step' S = state-of (do-cp-step (rough-state-of S))

```

```

typedef cdcl_W-state-inv-from-init-state = {S::cdcl_W-state-inv-st. cdcl_W-all-struct-inv (toS S)
   $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W\ (clauses\ (toS\ S))) (toS\ S)$ }

```

```

morphisms rough-state-from-init-state-of state-from-init-state-of

```

**proof**

```

show ( $([], [], [], 0, C\text{-True}) \in \{S. cdcl_W\text{-all-struct-inv } (toS\ S)$ 
   $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W\ (clauses\ (toS\ S))) (toS\ S)\}$ 
by (auto simp add: cdcl_W-all-struct-inv-def)

```

**qed**

**instantiation** *cdcl\_W-state-inv-from-init-state* :: *equal*

**begin**

**definition** *equal-cdcl\_W-state-inv-from-init-state* :: *cdcl\_W-state-inv-from-init-state*  $\Rightarrow$

```

cdcl_W-state-inv-from-init-state  $\Rightarrow$  bool where

```

```

equal-cdcl_W-state-inv-from-init-state S S'  $\longleftrightarrow$ 

```

```

(rough-state-from-init-state-of S = rough-state-from-init-state-of S')

```

**instance**

```

by standard (simp add: rough-state-from-init-state-of-inject

```

```

equal-cdcl_W-state-inv-from-init-state-def)

```

**end**

**definition** *ConI* **where**

```

ConI S = state-from-init-state-of (if cdcl_W-all-struct-inv (toS (fst S, snd S))

```

```

 $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W\ (clauses\ (toS\ S))) (toS\ S)$  then S else ([], [], [], 0, C-True))

```

**lemma** [*code abstype*]:

```

ConI (rough-state-from-init-state-of S) = S

```

```

using rough-state-from-init-state-of[of S] unfolding ConI-def by (simp add: rough-state-from-init-state-of-inverse)

```

**definition** *id-of-I-to::* *cdcl\_W-state-inv-from-init-state*  $\Rightarrow$  *cdcl\_W-state-inv* **where**

```

id-of-I-to S = state-of (rough-state-from-init-state-of S)

```

**lemma** [*code abstract*]:

*rough-state-of* (*id-of-I-to* *S*) = *rough-state-from-init-state-of* *S*  
**unfolding** *id-of-I-to-def* **using** *rough-state-from-init-state-of* **by** *auto*

**Conflict and Propagate** **function** *do-full1-cp-step* :: *cdcl<sub>W</sub>-state-inv*  $\Rightarrow$  *cdcl<sub>W</sub>-state-inv* **where**  
*do-full1-cp-step* *S* =

(*let* *S'* = *do-cp-step'* *S* *in*  
*if* *S* = *S'* *then* *S* *else* *do-full1-cp-step* *S'*)

**by** *auto*

**termination**

**proof** (*relation*  $\{(T', T). (rough-state-of\ T', rough-state-of\ T) \in \{(S', S).$

(*toS* *S'*, *toS* *S*)  $\in \{(S', S). cdcl_W-all-struct-inv\ S \wedge cdcl_W-cp\ S\ S'\}\}$ , *goal-cases*)

**case** 1

**show** ?*case*

**using** *wf-if-measure-f*[*OF* *wf-if-measure-f*[*OF* *cdcl<sub>W</sub>-cp-wf-all-inv*, *of toS*], *of rough-state-of*] .

**next**

**case** (2 *S'* *S*)

**then show** ?*case*

**unfolding** *do-cp-step'-def*

**apply** *simp*

**by** (*metis* *cp-step-is-cdcl<sub>W</sub>-cp* *rough-state-of-inverse*)

**qed**

**lemma** *do-full1-cp-step-fix-point-of-do-full1-cp-step*:

*do-cp-step*(*rough-state-of* (*do-full1-cp-step* *S*)) = (*rough-state-of* (*do-full1-cp-step* *S*))

**by** (*rule* *do-full1-cp-step.induct*[*of*  $\lambda S. do-cp-step(rough-state-of\ (do-full1-cp-step\ S))$

= (*rough-state-of* (*do-full1-cp-step* *S*))])

(*metis* (*full-types*) *do-full1-cp-step.elims* *rough-state-of-state-of-do-cp-step* *do-cp-step'-def*)

**lemma** *in-clauses-rough-state-of-is-distinct*:

*c*  $\in$  *set* (*clauses* (*rough-state-of* *S*) @ *learned-clss* (*rough-state-of* *S*))  $\implies$  *distinct* *c*

**apply** (*cases* *rough-state-of* *S*)

**using** *rough-state-of*[*of* *S*] **by** (*auto* *simp* *add*: *distinct-mset-set-distinct* *cdcl<sub>W</sub>-all-struct-inv-def* *distinct-cdcl<sub>W</sub>-state-def*)

**lemma** *do-full1-cp-step-full*:

*full* *cdcl<sub>W</sub>-cp* (*toS* (*rough-state-of* *S*))

(*toS* (*rough-state-of* (*do-full1-cp-step* *S*)))

**unfolding** *full-def* **apply** *standard*

**apply** (*induction* *S* *rule*: *do-full1-cp-step.induct*)

**apply** (*smt* *cp-step-is-cdcl<sub>W</sub>-cp* *do-cp-step'-def* *do-full1-cp-step.simps*

*rough-state-of-state-of-do-cp-step* *rtranclp.rtrancl-refl* *rtranclp-into-tranclp2*

*tranclp-into-rtranclp*)

**apply** (*rule* *do-cp-step-eq-no-step*[*OF* *do-full1-cp-step-fix-point-of-do-full1-cp-step*[*of* *S*]])

**using** *in-clauses-rough-state-of-is-distinct* **unfolding** *do-cp-step'-def* **by** *blast*

**lemma** [*code abstract*]:

*rough-state-of* (*do-cp-step'* *S*) = *do-cp-step* (*rough-state-of* *S*)

**unfolding** *do-cp-step'-def* **by** *auto*

**The other rules** **fun** *do-other-step* **where**

*do-other-step* *S* =

(*let* *T* = *do-skip-step* *S* *in*

*if* *T*  $\neq$  *S*

*then* *T*



```

else
  (let U = do-resolve-step T in
   if U ≠ T
   then U else
   (let V = do-backtrack-step U in
    if V ≠ U then V else do-decide-step V)))

```

**lemma** *do-other-step*:  
**assumes** *inv*: *cdcl<sub>W</sub>-all-struct-inv* (*toS S*) **and**  
*st*: *do-other-step S* ≠ *S*  
**shows** *cdcl<sub>W</sub>-o* (*toS S*) (*toS (do-other-step S)*)  
**using** *st inv* **by** (*auto split: split-if-asm*  
*simp add: Let-def*  
*intro: do-skip-step do-resolve-step do-backtrack-step do-decide-step*)

**lemma** *do-other-step-no*:  
**assumes** *inv*: *cdcl<sub>W</sub>-all-struct-inv* (*toS S*) **and**  
*st*: *do-other-step S* = *S*  
**shows** *no-step cdcl<sub>W</sub>-o* (*toS S*)  
**using** *st inv* **by** (*auto split: split-if-asm elim: cdcl<sub>W</sub>-bjE*  
*simp add: Let-def cdcl<sub>W</sub>-bj.simps elim!: cdcl<sub>W</sub>-o.cases*  
*dest!: do-skip-step-no do-resolve-step-no do-backtrack-step-no do-decide-step-no*)

**lemma** *rough-state-of-state-of-do-other-step[simp]*:  
*rough-state-of (state-of (do-other-step (rough-state-of S)))* = *do-other-step (rough-state-of S)*

**proof** (*cases do-other-step (rough-state-of S) = rough-state-of S*)

**case** *True*  
**then show** *?thesis* **by** *simp*

**next**

**case** *False*  
**have** *cdcl<sub>W</sub>-o* (*toS (rough-state-of S)*) (*toS (do-other-step (rough-state-of S))*)  
**by** (*metis False cdcl<sub>W</sub>-all-struct-inv-rough-state do-other-step[of rough-state-of S]*)  
**then have** *cdcl<sub>W</sub>-all-struct-inv* (*toS (do-other-step (rough-state-of S))*)  
**using** *cdcl<sub>W</sub>-all-struct-inv-inv cdcl<sub>W</sub>-all-struct-inv-rough-state other* **by** *blast*  
**then show** *?thesis*  
**by** (*simp add: CollectI state-of-inverse*)

**qed**

**definition** *do-other-step'* **where**

*do-other-step' S* =  
*state-of (do-other-step (rough-state-of S))*

**lemma** *rough-state-of-do-other-step'[code abstract]*:  
*rough-state-of (do-other-step' S)* = *do-other-step (rough-state-of S)*  
**apply** (*cases do-other-step (rough-state-of S) = rough-state-of S*)  
**unfolding** *do-other-step'-def* **apply** *simp*  
**using** *do-other-step[of rough-state-of S]* **by** (*smt cdcl<sub>W</sub>-all-struct-inv-inv*  
*cdcl<sub>W</sub>-all-struct-inv-rough-state mem-Collect-eq other state-of-inverse*)

**definition** *do-cdcl<sub>W</sub>-stgy-step* **where**

*do-cdcl<sub>W</sub>-stgy-step S* =  
(*let T = do-full1-cp-step S in*  
*if T ≠ S*  
*then T*  
*else*

(let  $U = (\text{do-other-step}' T)$  in  
 (do-full1-cp-step  $U$ )))

**definition**  $\text{do-cdcl}_W\text{-stgy-step}'$  **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**  $\text{toS-do-full1-cp-step-not-eq}$ :  $\text{do-full1-cp-step } S \neq S \implies$

$\text{toS } (\text{rough-state-of } S) \neq \text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } S))$

**proof** –

**assume**  $a1$ :  $\text{do-full1-cp-step } S \neq S$

**then have**  $S \neq \text{do-cp-step}' S$

**by** *fastforce*

**then show** *?thesis*

**by** (*metis* (*no-types*) *cp-step-is-cdcl<sub>W</sub>-cp do-cp-step'-def do-cp-step-eq-no-step*  
*do-full1-cp-step-fix-point-of-do-full1-cp-step in-clauses-rough-state-of-is-distinct*  
*rough-state-of-inverse*)

**qed**

$\text{do-full1-cp-step}$  should not be unfolded anymore:

**declare**  $\text{do-full1-cp-step.simps}$ [*simp del*]

**Correction of the transformation** **lemma**  $\text{do-cdcl}_W\text{-stgy-step}$ :

**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** (*cases*  $\text{do-full1-cp-step } S = S$ )

**case** *False*

**then show** *?thesis*

**using** *assms do-full1-cp-step-full*[*of S*] **unfolding** *full-unfold do-cdcl<sub>W</sub>-stgy-step-def*

**by** (*auto intro!*: *cdcl<sub>W</sub>-stgy.intros dest: toS-do-full1-cp-step-not-eq*)

**next**

**case** *True*

**have**  $\text{cdcl}_W\text{-o } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-other-step}' S)))$

**by** (*smt True assms cdcl<sub>W</sub>-all-struct-inv-rough-state do-cdcl<sub>W</sub>-stgy-step-def do-other-step*  
*rough-state-of-do-other-step' rough-state-of-inverse*)

**moreover**

**have**

*np*: *no-step propagate* ( $\text{toS } (\text{rough-state-of } S)$ ) **and**

*nc*: *no-step conflict* ( $\text{toS } (\text{rough-state-of } S)$ )

**apply** (*metis True do-cp-step-eq-no-prop-no-confl*

*do-full1-cp-step-fix-point-of-do-full1-cp-step do-propagate-step-no-step*  
*in-clauses-rough-state-of-is-distinct*)

**by** (*metis True do-conflict-step-no-step do-cp-step-eq-no-prop-no-confl*

*do-full1-cp-step-fix-point-of-do-full1-cp-step*)

**then have** *no-step cdcl<sub>W</sub>-cp* ( $\text{toS } (\text{rough-state-of } S)$ )

**by** (*simp add: cdcl<sub>W</sub>-cp.simps*)

**moreover have** *full cdcl<sub>W</sub>-cp* ( $\text{toS } (\text{rough-state-of } (\text{do-other-step}' S)))$

( $\text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } (\text{do-other-step}' S)))$ )

**using** *do-full1-cp-step-full* **by** *auto*

**ultimately show** *?thesis*

**using** *assms True unfolding do-cdcl<sub>W</sub>-stgy-step-def*

**by** (*auto intro!*: *cdcl<sub>W</sub>-stgy.other' dest: toS-do-full1-cp-step-not-eq*)

**qed**

**lemma**  $\text{length-trail-toS}$ [*simp*]:

$\text{length } (\text{trail } (\text{toS } S)) = \text{length } (\text{trail } S)$

```

by (cases S) auto

lemma conflicting-noTrue-iff-toS[simp]:
  conflicting (toS S)  $\neq$  C-True  $\longleftrightarrow$  conflicting S  $\neq$  C-True
by (cases S) auto

lemma trail-toS-neg-imp-trail-neg:
  trail (toS S)  $\neq$  trail (toS S')  $\implies$  trail S  $\neq$  trail S'
by (cases S, cases S') auto

lemma do-skip-step-trail-changed-or-conflict:
  assumes d: do-other-step S  $\neq$  S
  and inv: cdclW-all-struct-inv (toS S)
  shows trail S  $\neq$  trail (do-other-step S)
proof -
  have M:  $\bigwedge M K M1 c. M = c @ K \# M1 \implies \text{Suc} (\text{length } M1) \leq \text{length } M$ 
  by auto
  have cdclW-M-level-inv (toS S)
  using inv unfolding cdclW-all-struct-inv-def by auto
  have cdclW-o (toS S) (toS (do-other-step S)) using do-other-step[OF inv d] .
  then show ?thesis
  using  $\langle$ cdclW-M-level-inv (toS S) $\rangle$ 
  proof (induction toS (do-other-step S) rule: cdclW-o-induct-lev2)
    case decide
    then show ?thesis
    by (auto simp add: trail-toS-neg-imp-trail-neg)[]
  next
  case (skip)
  then show ?case
  by (cases S; cases do-other-step S) force
  next
  case (resolve)
  then show ?case
  by (cases S, cases do-other-step S) force
  next
  case (backtrack K i M1 M2 L D) note decomp = this(1) and confl-S = this(3) and undef =
  this(6) and
  U = this(7)
  have [simp]: cons-trail (Propagated L (D + {#L#}))
    (cdclW.reduce-trail-to M1
    (add-learned-cls (D + {#L#})
    (update-backtrack-lvl (get-maximum-level D (trail (toS S)))
    (update-conflicting C-True (toS S)))))
  =
  (Propagated L (D + {#L#})# M1, mset (map mset (clauses S)),
  {#D + {#L#}#} + mset (map mset (learned-cls S)),
  get-maximum-level D (trail (toS S)), C-True)
  apply (subst state-conv[of cons-trail -])
  using decomp undef by (cases S) auto
  then show ?case
  apply auto
  apply (cases do-other-step S; auto split: split-if-asm simp: Let-def)
  apply (cases S rule: do-skip-step.cases; auto split: split-if-asm)
  apply (cases S rule: do-skip-step.cases; auto split: split-if-asm)

```

```

    apply (cases S rule: do-backtrack-step.cases;
      auto split: split-if-asm option.splits list.splits marked-lit.splits
      dest!: bt-cut-some-decomp)[]
  using d apply (cases S rule: do-decide-step.cases; auto split: option.splits)[]
done
qed
qed

lemma do-full1-cp-step-induct:
  ( $\bigwedge S. (S \neq \text{do-cp-step}' S \implies P (\text{do-cp-step}' S)) \implies P S$ )  $\implies P a0$ 
  using do-full1-cp-step.induct by metis

lemma do-cp-step-neq-trail-increase:
   $\exists c. \text{trail} (\text{do-cp-step } S) = c @ \text{trail } S \wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$ 
  by (cases S, cases conflicting S)
    (auto simp add: do-cp-step-def do-conflict-step-def do-propagate-step-def split: option.splits)

lemma do-full1-cp-step-neq-trail-increase:
   $\exists c. \text{trail} (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{trail} (\text{rough-state-of } S)$ 
   $\wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$ 
  apply (induction rule: do-full1-cp-step-induct)
  apply (case-tac do-cp-step' S = S)
  apply (simp add: do-full1-cp-step.simps)
  by (smt Un-iff append-assoc do-cp-step'-def do-cp-step-neq-trail-increase do-full1-cp-step.simps
    rough-state-of-state-of-do-cp-step set-append)

lemma do-cp-step-conflicting:
   $\text{conflicting} (\text{rough-state-of } S) \neq C\text{-True} \implies \text{do-cp-step}' S = S$ 
  unfolding do-cp-step'-def do-cp-step-def by simp

lemma do-full1-cp-step-conflicting:
   $\text{conflicting} (\text{rough-state-of } S) \neq C\text{-True} \implies \text{do-full1-cp-step } S = S$ 
  unfolding do-cp-step'-def do-cp-step-def
  apply (induction rule: do-full1-cp-step-induct)
  by (case-tac S  $\neq$  do-cp-step' S)
    (auto simp add: rough-state-of-inverse do-full1-cp-step.simps dest: do-cp-step-conflicting)

lemma do-decide-step-not-conflicting-one-more-decide:
  assumes
    conflicting S = C-True and
    do-decide-step S  $\neq$  S
  shows Suc (length (filter is-marked (trail S)))
    = length (filter is-marked (trail (do-decide-step S)))
  using assms unfolding do-other-step'-def
  by (cases S) (auto simp: Let-def split: split-if-asm option.splits
    dest!: find-first-unused-var-Some-not-all-incl)

lemma do-decide-step-not-conflicting-one-more-decide-bt:
  assumes conflicting S  $\neq$  C-True and
    do-decide-step S  $\neq$  S
  shows length (filter is-marked (trail S)) < length (filter is-marked (trail (do-decide-step S)))
  using assms unfolding do-other-step'-def by (cases S, cases conflicting S)
    (auto simp add: Let-def split: split-if-asm option.splits)

lemma do-other-step-not-conflicting-one-more-decide-bt:

```

**assumes** *conflicting* (*rough-state-of*  $S$ )  $\neq$   $C\text{-True}$  **and**  
*conflicting* (*rough-state-of* (*do-other-step'*  $S$ )) =  $C\text{-True}$  **and**  
*do-other-step'*  $S \neq S$   
**shows** *length* (*filter is-marked* (*trail* (*rough-state-of*  $S$ )))  
 $>$  *length* (*filter is-marked* (*trail* (*rough-state-of* (*do-other-step'*  $S$ ))))  
**proof** (*cases*  $S$ , *goal-cases*)  
**case** ( $1\ y$ ) **note**  $S = \text{this}(1)$  **and**  $\text{inv} = \text{this}(2)$   
**obtain**  $M\ N\ U\ k\ E$  **where**  $y: y = (M, N, U, k, C\text{-Clause } E)$   
**using** *assms*( $1$ )  $S\ \text{inv}$  **by** (*cases*  $y$ , *cases conflicting*  $y$ ) *auto*  
**have**  $M$ : *rough-state-of* (*state-of* ( $M, N, U, k, C\text{-Clause } E$ )) = ( $M, N, U, k, C\text{-Clause } E$ )  
**using**  $\text{inv } y$  **by** (*auto simp add: state-of-inverse*)  
**have**  $bt$ : *do-other-step'*  $S = \text{state-of}$  (*do-backtrack-step* (*rough-state-of*  $S$ ))  
  
**using** *assms*( $1, 2$ ) **apply** (*cases rough-state-of* (*do-other-step'*  $S$ ))  
**apply** (*auto simp add: Let-def do-other-step'-def*)  
**apply** (*cases rough-state-of*  $S$  *rule: do-decide-step.cases*)  
**apply** *auto*  
**done**  
**show** *?case*  
**using** *assms*( $2$ )  $S$  **unfolding**  $bt\ y\ \text{inv}$   
**apply** *simp*  
**by** (*auto simp add: M*  
*split: option.splits*  
*dest: bt-cut-some-decomp arg-cong[of - -  $\lambda u. \text{length}(\text{filter is-marked } u)$ ])  
**qed***

**lemma** *do-other-step-not-conflicting-one-more-decide*:  
**assumes** *conflicting* (*rough-state-of*  $S$ ) =  $C\text{-True}$  **and**  
*do-other-step'*  $S \neq S$   
**shows**  $1 + \text{length}(\text{filter is-marked}(\text{trail}(\text{rough-state-of } S)))$   
 $= \text{length}(\text{filter is-marked}(\text{trail}(\text{rough-state-of}(\text{do-other-step}' S))))$   
**proof** (*cases*  $S$ , *goal-cases*)  
**case** ( $1\ y$ ) **note**  $S = \text{this}(1)$  **and**  $\text{inv} = \text{this}(2)$   
**obtain**  $M\ N\ U\ k$  **where**  $y: y = (M, N, U, k, C\text{-True})$  **using** *assms*( $1$ )  $S\ \text{inv}$  **by** (*cases*  $y$ ) *auto*  
**have**  $M$ : *rough-state-of* (*state-of* ( $M, N, U, k, C\text{-True}$ )) = ( $M, N, U, k, C\text{-True}$ )  
**using**  $\text{inv } y$  **by** (*auto simp add: state-of-inverse*)  
**have** *state-of* (*do-decide-step* ( $M, N, U, k, C\text{-True}$ ))  $\neq$  *state-of* ( $M, N, U, k, C\text{-True}$ )  
**using** *assms*( $2$ ) **unfolding** *do-other-step'-def*  $y\ \text{inv } S$  **by** (*auto simp add: M*)  
**then have**  $f4$ : *do-skip-step* (*rough-state-of*  $S$ ) = *rough-state-of*  $S$   
**unfolding**  $S\ M\ y$  **by** (*metis* (*full-types*) *do-skip-step.simps*( $4$ ))  
**have**  $f5$ : *do-resolve-step* (*rough-state-of*  $S$ ) = *rough-state-of*  $S$   
**unfolding**  $S\ M\ y$  **by** (*metis* (*no-types*) *do-resolve-step.simps*( $4$ ))  
**have**  $f6$ : *do-backtrack-step* (*rough-state-of*  $S$ ) = *rough-state-of*  $S$   
**unfolding**  $S\ M\ y$  **by** (*metis* (*no-types*) *do-backtrack-step.simps*( $2$ ))  
**have** *do-other-step* (*rough-state-of*  $S$ )  $\neq$  *rough-state-of*  $S$   
**using** *assms*( $2$ ) **unfolding**  $S\ M\ y$  *do-other-step'-def* **by** (*metis* (*no-types*))  
**then show** *?case*  
**using**  $f6\ f5\ f4$  **by** (*simp add: assms*( $1$ ) *do-decide-step-not-conflicting-one-more-decide*  
*do-other-step'-def*)  
**qed**

**lemma** *rough-state-of-state-of-do-skip-step-rough-state-of[simp]*:  
*rough-state-of* (*state-of* (*do-skip-step* (*rough-state-of*  $S$ ))) = *do-skip-step* (*rough-state-of*  $S$ )  
**by** (*smt do-other-step.simps rough-state-of-inverse rough-state-of-state-of-do-other-step*)

**lemma** *conflicting-do-resolve-step-iff*[iff]:  
 $\text{conflicting } (\text{do-resolve-step } S) = C\text{-True} \longleftrightarrow \text{conflicting } S = C\text{-True}$   
**by** (cases  $S$  rule: *do-resolve-step.cases*)  
(auto simp add: *Let-def split: option.splits*)

**lemma** *conflicting-do-skip-step-iff*[iff]:  
 $\text{conflicting } (\text{do-skip-step } S) = C\text{-True} \longleftrightarrow \text{conflicting } S = C\text{-True}$   
**by** (cases  $S$  rule: *do-skip-step.cases*)  
(auto simp add: *Let-def split: option.splits*)

**lemma** *conflicting-do-decide-step-iff*[iff]:  
 $\text{conflicting } (\text{do-decide-step } S) = C\text{-True} \longleftrightarrow \text{conflicting } S = C\text{-True}$   
**by** (cases  $S$  rule: *do-decide-step.cases*)  
(auto simp add: *Let-def split: option.splits*)

**lemma** *conflicting-do-backtrack-step-imp*[simp]:  
 $\text{do-backtrack-step } S \neq S \implies \text{conflicting } (\text{do-backtrack-step } S) = C\text{-True}$   
**by** (cases  $S$  rule: *do-backtrack-step.cases*)  
(auto simp add: *Let-def split: list.splits option.splits marked-lit.splits*)

**lemma** *do-skip-step-eq-iff-trail-eq*:  
 $\text{do-skip-step } S = S \longleftrightarrow \text{trail } (\text{do-skip-step } S) = \text{trail } S$   
**by** (cases  $S$  rule: *do-skip-step.cases*) auto

**lemma** *do-decide-step-eq-iff-trail-eq*:  
 $\text{do-decide-step } S = S \longleftrightarrow \text{trail } (\text{do-decide-step } S) = \text{trail } S$   
**by** (cases  $S$  rule: *do-decide-step.cases*) (auto split: *option.split*)

**lemma** *do-backtrack-step-eq-iff-trail-eq*:  
 $\text{do-backtrack-step } S = S \longleftrightarrow \text{trail } (\text{do-backtrack-step } S) = \text{trail } S$   
**by** (cases  $S$  rule: *do-backtrack-step.cases*)  
(auto split: *option.split list.splits marked-lit.splits*  
dest!: *bt-cut-in-get-all-marked-decomposition*)

**lemma** *do-resolve-step-eq-iff-trail-eq*:  
 $\text{do-resolve-step } S = S \longleftrightarrow \text{trail } (\text{do-resolve-step } S) = \text{trail } S$   
**by** (cases  $S$  rule: *do-resolve-step.cases*) auto

**lemma** *do-other-step-eq-iff-trail-eq*:  
 $\text{trail } (\text{do-other-step } S) = \text{trail } S \longleftrightarrow \text{do-other-step } S = S$   
**by** (auto simp add: *Let-def do-skip-step-eq-iff-trail-eq[symmetric]*  
*do-decide-step-eq-iff-trail-eq[symmetric]* *do-backtrack-step-eq-iff-trail-eq[symmetric]*  
*do-resolve-step-eq-iff-trail-eq[symmetric]*)

**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**  $\text{do-other-step}' S = S \wedge \text{do-full1-cp-step } S = S$   
**proof** –  
let  $?T = \text{do-other-step}' S$   
{ **assume** *conf!*:  $\text{conflicting } (\text{rough-state-of } ?T) \neq C\text{-True}$   
**then have**  $tr$ :  $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } ?T)) = \text{trail } (\text{rough-state-of } ?T)$   
**using** *do-full1-cp-step-conflicting* **by** auto  
**have**  $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } (\text{do-other-step}' S))) = \text{trail } (\text{rough-state-of } S)$

```

    using arg-cong[OF H, of  $\lambda S. \text{trail} (\text{rough-state-of } S)$ ] .
  then have  $\text{trail} (\text{rough-state-of } (\text{do-other-step}' S)) = \text{trail} (\text{rough-state-of } S)$ 
    by (auto simp add: do-full1-cp-step-conflicting confl)
  then have  $\text{do-other-step}' S = S$ 
    by (simp add: do-other-step-eq-iff-trail-eq do-other-step'-def rough-state-of-inverse
      del: do-other-step.simps)
}
moreover {
  assume eq[simp]:  $\text{do-other-step}' S = S$ 
  obtain c where c:  $\text{trail} (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{trail} (\text{rough-state-of } S)$ 
    using do-full1-cp-step-neq-trail-increase by auto

  moreover have  $\text{trail} (\text{rough-state-of } (\text{do-full1-cp-step } S)) = \text{trail} (\text{rough-state-of } S)$ 
    using arg-cong[OF H, of  $\lambda S. \text{trail} (\text{rough-state-of } S)$ ] by simp
  finally have  $c = []$  by blast
  then have  $\text{do-full1-cp-step } S = S$  using assms by auto
}
moreover {
  assume confl:  $\text{conflicting} (\text{rough-state-of } ?T) = C\text{-True}$  and neg:  $\text{do-other-step}' S \neq S$ 
  obtain c where
    c:  $\text{trail} (\text{rough-state-of } (\text{do-full1-cp-step } ?T)) = c @ \text{trail} (\text{rough-state-of } ?T)$  and
    nm:  $\forall m \in \text{set } c. \neg \text{is-marked } m$ 
    using do-full1-cp-step-neq-trail-increase by auto
  have  $\text{length} (\text{filter is-marked } (\text{trail} (\text{rough-state-of } (\text{do-full1-cp-step } ?T))))$ 
    =  $\text{length} (\text{filter is-marked } (\text{trail} (\text{rough-state-of } ?T)))$  using nm unfolding c by force
  moreover have  $\text{length} (\text{filter is-marked } (\text{trail} (\text{rough-state-of } S)))$ 
     $\neq \text{length} (\text{filter is-marked } (\text{trail} (\text{rough-state-of } ?T)))$ 
    using do-other-step-not-conflicting-one-more-decide[OF - neg]
    do-other-step-not-conflicting-one-more-decide-bt[of S, OF - confl neg]
    by linarith
  finally have False unfolding H by blast
}
ultimately show ?thesis by blast
qed

```

**lemma** *do-cdcl<sub>W</sub>-stgy-step-no*:

```

  assumes S:  $\text{do-cdcl}_W\text{-stgy-step } S = S$ 
  shows no-step cdclW-stgy (toS (rough-state-of S))
proof -
  {
    fix S'
    assume full1  $\text{cdcl}_W\text{-cp } (\text{toS } (\text{rough-state-of } S)) S'$ 
    then have False
      using do-full1-cp-step-full[of S] unfolding full-def S rtranclp-unfold full1-def
      by (smt assms do-cdclW-stgy-step-def tranclpD)
  }
  moreover {
    fix S' S''
    assume  $\text{cdcl}_W\text{-o } (\text{toS } (\text{rough-state-of } S)) S'$  and
      no-step propagate  $(\text{toS } (\text{rough-state-of } S))$  and
      no-step conflict  $(\text{toS } (\text{rough-state-of } S))$  and
      full  $\text{cdcl}_W\text{-cp } S' S''$ 
    then have False
      using assms unfolding do-cdclW-stgy-step-def
      by (smt cdclW-all-struct-inv-rough-state do-full1-cp-step-do-other-step'-normal-form)
  }

```

```

    do-other-step-no rough-state-of-do-other-step')
  }
  ultimately show ?thesis using assms by (force simp: cdclW-cp.simps cdclW-stgy.simps)
qed

lemma toS-rough-state-of-state-of-rough-state-from-init-state-of[simp]:
  toS (rough-state-of (state-of (rough-state-from-init-state-of S)))
    = toS (rough-state-from-init-state-of S)
  using rough-state-from-init-state-of[of S] by (auto simp add: state-of-inverse)

lemma cdclW-cp-is-rtrancp-cdclW: cdclW-cp S T  $\implies$  cdclW** S T
  apply (induction rule: cdclW-cp.induct)
  using conflict apply blast
  using propagate by blast

lemma rtrancp-cdclW-cp-is-rtrancp-cdclW: cdclW-cp** S T  $\implies$  cdclW** S T
  apply (induction rule: rtrancp-induct)
  apply simp
  by (fastforce dest!: cdclW-cp-is-rtrancp-cdclW)

lemma cdclW-stgy-is-rtrancp-cdclW:
  cdclW-stgy S T  $\implies$  cdclW** S T
  apply (induction rule: cdclW-stgy.induct)
  using cdclW-stgy.conflict' rtrancp-cdclW-stgy-rtrancp-cdclW apply blast
  unfolding full-def by (fastforce dest!:cdclW.other rtrancp-cdclW-cp-is-rtrancp-cdclW)

lemma cdclW-stgy-init-clss: cdclW-stgy S T  $\implies$  cdclW-M-level-inv S  $\implies$  clauses S = clauses T
  using rtrancp-cdclW-init-clss cdclW-stgy-is-rtrancp-cdclW by fast

lemma clauses-toS-rough-state-of-do-cdclW-stgy-step[simp]:
  clauses (toS (rough-state-of (do-cdclW-stgy-step (state-of (rough-state-from-init-state-of S)))))
    = clauses (toS (rough-state-from-init-state-of S)) (is - = clauses (toS ?S))
  apply (cases do-cdclW-stgy-step (state-of ?S) = state-of ?S)
  apply simp
  by (smt cdclW-all-struct-inv-def cdclW-all-struct-inv-rough-state cdclW-stgy-no-more-init-clss
    do-cdclW-stgy-step toS-rough-state-of-state-of-rough-state-from-init-state-of)

lemma rough-state-from-init-state-of-do-cdclW-stgy-step'[code abstract]:
  rough-state-from-init-state-of (do-cdclW-stgy-step' S) =
    rough-state-of (do-cdclW-stgy-step (id-of-I-to S))
proof -
  let ?S = (rough-state-from-init-state-of S)
  have cdclW-stgy** (S0-cdclW (clauses (toS (rough-state-from-init-state-of S))))
    (toS (rough-state-from-init-state-of S))
    using rough-state-from-init-state-of[of S] by auto
  moreover have cdclW-stgy**
    (toS (rough-state-from-init-state-of S))
    (toS (rough-state-of (do-cdclW-stgy-step
      (state-of (rough-state-from-init-state-of S)))))
    using do-cdclW-stgy-step[of state-of ?S]
    by (cases do-cdclW-stgy-step (state-of ?S) = state-of ?S) auto
  ultimately show ?thesis
    unfolding do-cdclW-stgy-step'-def id-of-I-to-def by (auto intro!: state-from-init-state-of-inverse)
qed

```



**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*)

by *fast+*

**termination**

**proof** (relation {(*T*, *S*).

(*cdcl<sub>W</sub>-measure* (*toS* (*rough-state-from-init-state-of* *T*))),

*cdcl<sub>W</sub>-measure* (*toS* (*rough-state-from-init-state-of* *S*))))

∈ *lexn* {(*a*, *b*). *a* < *b*} *3*}, *goal-cases*)

**case** 1

**show** ?*case* by (rule *wf-if-measure-f*) (*auto intro!*: *wf-lexn wf-less*)

**next**

**case** (2 *S T*) **note** *T* = *this*(1) **and** *ST* = *this*(2)

let ?*S* = *rough-state-from-init-state-of* *S*

**have** *S*: *cdcl<sub>W</sub>-stgy*\*\* (*S0-cdcl<sub>W</sub>* (*clauses* (*toS* ?*S*))) (*toS* ?*S*)

**using** *rough-state-from-init-state-of*[*of S*] **by** *auto*

**moreover have** *cdcl<sub>W</sub>-stgy* (*toS* (*rough-state-from-init-state-of* *S*))

(*toS* (*rough-state-from-init-state-of* *T*))

**using** *ST do-cdcl<sub>W</sub>-stgy-step unfolding T*

**by** (*smt id-of-I-to-def mem-Collect-eq rough-state-from-init-state-of*  
*rough-state-from-init-state-of-do-cdcl<sub>W</sub>-stgy-step'* *rough-state-from-init-state-of-inject*  
*state-of-inverse*)

**moreover**

**have** *cdcl<sub>W</sub>-all-struct-inv* (*toS* (*rough-state-from-init-state-of* *S*))

**using** *rough-state-from-init-state-of*[*of S*] **by** *auto*

**then have** *cdcl<sub>W</sub>-all-struct-inv* (*S0-cdcl<sub>W</sub>* (*clauses* (*toS* (*rough-state-from-init-state-of* *S*))))

**by** (*cases rough-state-from-init-state-of S*)

(*auto simp add: cdcl<sub>W</sub>-all-struct-inv-def distinct-cdcl<sub>W</sub>-state-def*)

**ultimately show** ?*case*

**by** (*auto intro!*: *cdcl<sub>W</sub>-stgy-step-decreasing*[*of - - S0-cdcl<sub>W</sub>* (*clauses* (*toS* ?*S*))]

*simp del: cdcl<sub>W</sub>-measure.simps*)

**qed**

**thm** *do-all-cdcl<sub>W</sub>-stgy.induct*

**lemma** *do-all-cdcl<sub>W</sub>-stgy-induct*:

( $\bigwedge S. (\text{do-cdcl}_W\text{-stgy-step}' S \neq S \implies P (\text{do-cdcl}_W\text{-stgy-step}' S)) \implies P S) \implies P a0$

**using** *do-all-cdcl<sub>W</sub>-stgy.induct* **by** *metis*

**lemma** *no-step-cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-all*:

*no-step cdcl<sub>W</sub>-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl<sub>W</sub>-stgy* *S*))))

**apply** (*induction S rule:do-all-cdcl<sub>W</sub>-stgy-induct*)

**apply** (*case-tac do-cdcl<sub>W</sub>-stgy-step' S ≠ S*)

**proof** –

**fix** *Sa* :: *cdcl<sub>W</sub>-state-inv-from-init-state*

**assume** *a1*:  $\neg \text{do-cdcl}_W\text{-stgy-step}' Sa \neq Sa$

{ **fix** *pp*

**have** (if *True* then *Sa* else *do-all-cdcl<sub>W</sub>-stgy* *Sa*) = *do-all-cdcl<sub>W</sub>-stgy* *Sa*

**using** *a1* **by** *auto*

**then have**  $\neg \text{cdcl}_W\text{-stgy}$  (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl<sub>W</sub>-stgy* *Sa*)))) *pp*

**using** *a1* **by** (*metis* (*no-types*) *do-cdcl<sub>W</sub>-stgy-step-no id-of-I-to-def*

*rough-state-from-init-state-of-do-cdcl<sub>W</sub>-stgy-step'* *rough-state-of-inverse*) }

**then show** *no-step cdcl<sub>W</sub>-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl<sub>W</sub>-stgy* *Sa*))))

**by** *fastforce*

**next**

```

fix Sa :: cdclW-state-inv-from-init-state
assume a1: do-cdclW-stgy-step' Sa ≠ Sa
  ⇒ no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy (do-cdclW-stgy-step' Sa))))
assume a2: do-cdclW-stgy-step' Sa ≠ Sa
have do-all-cdclW-stgy Sa = do-all-cdclW-stgy (do-cdclW-stgy-step' Sa)
  by (metis (full-types) do-all-cdclW-stgy.simps)
then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
  using a2 a1 by presburger
qed

```

```

lemma do-all-cdclW-stgy-is-rtrancpl-cdclW-stgy:
  cdclW-stgy** (toS (rough-state-from-init-state-of S))
    (toS (rough-state-from-init-state-of (do-all-cdclW-stgy S)))
  apply (induction S rule: do-all-cdclW-stgy-induct)
  apply (case-tac do-cdclW-stgy-step' S = S)
  apply simp
  by (smt converse-rtrancpl-into-rtrancpl do-all-cdclW-stgy.simps do-cdclW-stgy-step id-of-I-to-def
    rough-state-from-init-state-of-do-cdclW-stgy-step'
    toS-rough-state-of-state-of-rough-state-from-init-state-of)

```

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, C-True)))) = *S* **and**

*S*: (*M'*, *N'*, *U'*, *k*, *E*) = toS *S*

**shows** (*E* ≠ *C*-Clause {#} ∧ satisfiable (set (map mset *N*)))  
 ∨ (*E* = *C*-Clause {#} ∧ unsatisfiable (set (map mset *N*)))

**proof** –

let ?*N* = map remdups *N*

**have** *inv*: cdcl<sub>W</sub>-all-struct-inv (toS ([], map remdups *N*, [], 0, C-True))

**unfolding** cdcl<sub>W</sub>-all-struct-inv-def distinct-cdcl<sub>W</sub>-state-def distinct-mset-set-def **by** auto

**then have** *S0*: rough-state-of (state-of ([], map remdups *N*, [], 0, C-True))

= ([], map remdups *N*, [], 0, C-True) **by** simp

**have** *1*: full cdcl<sub>W</sub>-stgy (toS ([], ?*N*, [], 0, C-True)) (toS *S*)

**unfolding** full-def **apply** rule

**using** do-all-cdcl<sub>W</sub>-stgy-is-rtrancpl-cdcl<sub>W</sub>-stgy[*of*  
 state-from-init-state-of ([], map remdups *N*, [], 0, C-True)] *inv*  
 no-step-cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-all

**by** (auto simp del: do-all-cdcl<sub>W</sub>-stgy.simps simp: state-from-init-state-of-inverse  
 r[symmetric])+

**moreover have** *2*: finite (set (map mset ?*N*)) **by** auto

**moreover have** *3*: distinct-mset-set (set (map mset ?*N*))

**unfolding** distinct-mset-set-def **by** auto

**moreover**

**have** cdcl<sub>W</sub>-all-struct-inv (toS *S*)

**by** (metis (no-types) cdcl<sub>W</sub>-all-struct-inv-rough-state *r*  
 toS-rough-state-of-state-of-rough-state-from-init-state-of)

**then have** *cons*: consistent-interp (lits-of *M'*)

**unfolding** cdcl<sub>W</sub>-all-struct-inv-def cdcl<sub>W</sub>-*M*-level-inv-def *S*[symmetric] **by** auto

**moreover**

**have** *clauses* (toS ([], ?*N*, [], 0, C-True)) = *clauses* (toS *S*)

**apply** (rule rtrancpl-cdcl<sub>W</sub>-init-clss)

**using** *1* **unfolding** full-def **by** (auto simp add: rtrancpl-cdcl<sub>W</sub>-stgy-rtrancpl-cdcl<sub>W</sub>)

```

    then have  $N'$ :  $mset (map mset ?N) = N'$ 
      using  $S[symmetric]$  by auto
  have  $(E \neq C\text{-Clause } \{\#\} \wedge \text{satisfiable } (set (map mset ?N)))$ 
     $\vee (E = C\text{-Clause } \{\#\} \wedge \text{unsatisfiable } (set (map mset ?N)))$ 
    using full-cdclW-stgy-final-state-conclusive unfolding  $N'$  apply rule
      using 1 apply simp
      using 2 apply simp
      using 3 apply simp
      using  $S[symmetric]$   $N'$  apply auto[1]
    using  $S[symmetric]$   $N'$  cons by (fastforce simp: true-annots-true-cls)
  then show ?thesis by auto
qed

```

**The Code** The SML code is skipped in the documentation, but stays to ensure that some version of the exported code is working

```

end
theory CDCL-WNOT
imports CDCL-W-Termination CDCL-NOT
begin

```

## 19 Link between Weidenbach's and NOT's CDCL

### 19.1 Inclusion of the states

```

declare upt.simps(2)[simp del]
sledgehammer-params[verbose]

```

```

context cdclW-ops
begin

```

**lemma** *backtrack-levE*:

```

  backtrack  $S S' \implies \text{cdcl}_W\text{-}M\text{-level-inv } S \implies
  (\bigwedge D L K M1 M2.
    (Marked K (Suc (get-maximum-level D (trail S))) \# M1, M2)
      \in set (get-all-marked-decomposition (trail S)) \implies
    get-level L (trail S) = get-maximum-level (D + \{\#L\# \}) (trail S) \implies
    undefined-lit M1 L \implies
     $S' \sim \text{cons-trail } (Propagated L (D + \{\#L\# \}))
      (reduce-trail-to M1 (add-learned-cls (D + \{\#L\# \}))
        (update-backtrack-lvl (get-maximum-level D (trail S)) (update-conflicting C\text{-True } S))) \implies
    \text{backtrack-lvl } S = \text{get-maximum-level } (D + \{\#L\# \}) (trail S) \implies
    \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \}) \implies P \implies
  P
  using assms by (induction rule: backtrack-induction-lev2) metis$$ 
```

**lemma** *backtrack-no-cdcl<sub>W</sub>-bj*:

```

  assumes cdcl: cdclW-bj  $T U$  and inv: cdclW-M-level-inv  $V$ 
  shows  $\neg \text{backtrack } V T$ 
  using cdcl inv
  by (induction rule: cdclW-bj.induct) (force elim!: backtrack-levE[OF - inv]
    simp: cdclW-M-level-inv-def) +

```

**abbreviation** *skip-or-resolve* ::  $'st \Rightarrow 'st \Rightarrow \text{bool}$  **where**

$skip\text{-}or\text{-}resolve \equiv (\lambda S T. skip\ S\ T \vee resolve\ S\ T)$

**lemma** *rtranclp-cdcl<sub>W</sub>-bj-skip-or-resolve-backtrack*:  
**assumes** *cdcl<sub>W</sub>-bj<sup>\*\*</sup> S U* **and** *inv: cdcl<sub>W</sub>-M-level-inv S*  
**shows** *skip-or-resolve<sup>\*\*</sup> S U*  $\vee$   $(\exists T. skip\text{-}or\text{-}resolve^{**}\ S\ T \wedge backtrack\ T\ U)$   
**using** *assms*  
**proof** (*induction*)  
**case** *base*  
**then show** *?case* **by** *simp*  
**next**  
**case** (*step U V*) **note** *st = this(1)* **and** *bj = this(2)* **and** *IH = this(3)[OF this(4)]*  
**consider**  
 (*SU*) *S = U*  
 | (*SUp*) *cdcl<sub>W</sub>-bj<sup>++</sup> S U*  
**using** *st* **unfolding** *rtranclp-unfold* **by** *blast*  
**then show** *?case*  
**proof** *cases*  
**case** *SUp*  
**have**  $\bigwedge T. skip\text{-}or\text{-}resolve^{**}\ S\ T \implies cdcl_W^{**}\ S\ T$   
**using** *mono-rtranclp[of skip-or-resolve cdcl<sub>W</sub>]* *other* **by** *blast*  
**then have** *skip-or-resolve<sup>\*\*</sup> S U*  
**using** *bj IH inv backtrack-no-cdcl<sub>W</sub>-bj rtranclp-cdcl<sub>W</sub>-consistent-inv[OF - inv]* **by** *meson*  
**then show** *?thesis*  
**using** *bj* **by** (*metis (no-types, lifting) cdcl<sub>W</sub>-bj.cases rtranclp.simps*)  
**next**  
**case** *SU*  
**then show** *?thesis*  
**using** *bj* **by** (*metis (no-types, lifting) cdcl<sub>W</sub>-bj.cases rtranclp.simps*)  
**qed**  
**qed**

**lemma** *rtranclp-skip-or-resolve-rtranclp-cdcl<sub>W</sub>*:  
*skip-or-resolve<sup>\*\*</sup> S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*  
**by** (*induction rule: rtranclp-induct*) (*auto dest!: cdcl<sub>W</sub>-bj.intros cdcl<sub>W</sub>.intros cdcl<sub>W</sub>-o.intros*)

**abbreviation** *backjump-l-cond* :: '*v literal multiset*  $\Rightarrow$  '*v literal*  $\Rightarrow$  '*st*  $\Rightarrow$  *bool* **where**  
*backjump-l-cond*  $\equiv \lambda C\ L\ S. True$

**definition** *inv<sub>NOT</sub>* :: '*st*  $\Rightarrow$  *bool* **where**  
*inv<sub>NOT</sub>*  $\equiv \lambda S. no\text{-}dup\ (trail\ S)$

**declare** *inv<sub>NOT</sub>-def[simp]*  
**end**

**fun** *convert-trail-from-W* ::  
 ('*v*, '*lvl*, '*v literal multiset*) *marked-lit list*  
 $\Rightarrow$  ('*v*, *unit*, *unit*) *marked-lit list* **where**  
*convert-trail-from-W* [] = [] |  
*convert-trail-from-W* (*Propagated L* - # *M*) = *Propagated L* () # *convert-trail-from-W M* |  
*convert-trail-from-W* (*Marked L* - # *M*) = *Marked L* () # *convert-trail-from-W M*

**lemma** *atm-convert-trail-from-W[simp]*:  
 $(\lambda l. atm\text{-}of\ (lit\text{-}of\ l))\ 'set\ (convert\text{-}trail\text{-}from\text{-}W\ xs) = (\lambda l. atm\text{-}of\ (lit\text{-}of\ l))\ 'set\ xs$   
**by** (*induction rule: marked-lit-list-induct*) *simp-all*

**lemma** *no-dup-convert-from-W*[simp]:  
*no-dup* (convert-trail-from-W M)  $\longleftrightarrow$  *no-dup* M  
**by** (induction rule: marked-lit-list-induct) simp-all

**lemma** *lits-of-convert-trail-from-W*[simp]:  
*lits-of* (convert-trail-from-W M) = *lits-of* M  
**by** (induction rule: marked-lit-list-induct) simp-all

**lemma** *convert-trail-from-W-true-annots*[simp]:  
convert-trail-from-W M  $\models_{as}$  C  $\longleftrightarrow$  M  $\models_{as}$  C  
**by** (auto simp: true-annots-true-cls)

**lemma** *defined-lit-convert-trail-from-W*[simp]:  
defined-lit (convert-trail-from-W S) L  $\longleftrightarrow$  defined-lit S L  
**by** (auto simp: defined-lit-map)

**lemma** *convert-trail-from-W-append*[simp]:  
convert-trail-from-W (M @ M') = convert-trail-from-W M @ convert-trail-from-W M'  
**by** (induction M rule: marked-lit-list-induct) simp-all

**lemma** *length-convert-trail-from-W*[simp]:  
length (convert-trail-from-W W) = length W  
**by** (induction W rule: convert-trail-from-W.induct) auto

**lemma** *convert-trail-from-W-nil-iff*[simp]: convert-trail-from-W S = []  $\longleftrightarrow$  S = []  
**by** (induction S rule: convert-trail-from-W.induct) auto

The values 0 and {#} do not matter.

**fun** *convert-marked-lit-from-NOT* **where**  
convert-marked-lit-from-NOT (Propagated L -) = Propagated L {#} |  
convert-marked-lit-from-NOT (Marked L -) = Marked L 0

**fun** *convert-trail-from-NOT* ::  
('v, unit, unit) marked-lit list  
 $\Rightarrow$  ('v, nat, 'v literal multiset) marked-lit list **where**  
convert-trail-from-NOT [] = [] |  
convert-trail-from-NOT (L # M) = convert-marked-lit-from-NOT L # convert-trail-from-NOT M

**lemma** *convert-trail-from-W-from-NOT*[simp]:  
convert-trail-from-W (convert-trail-from-NOT M) = M  
**by** (induction rule: marked-lit-list-induct) auto

**lemma** *convert-trail-from-W-cons-convert-lit-from-NOT*[simp]:  
convert-trail-from-W (convert-marked-lit-from-NOT L # M) = L # convert-trail-from-W M  
**by** (cases L) auto

**lemma** *convert-trail-from-W-tl*[simp]:  
convert-trail-from-W (tl M) = tl (convert-trail-from-W M)  
**by** (induction rule: convert-trail-from-W.induct) simp-all

**lemma** *length-convert-trail-from-NOT*[simp]:  
length (convert-trail-from-NOT W) = length W  
**by** (induction W rule: convert-trail-from-NOT.induct) auto

**abbreviation** *trail<sub>NOT</sub>* **where**

$trail_{NOT} \equiv convert-trail-from-W \circ fst$

**lemma** *undefined-lit-convert-trail-from-W*[iff]:  
 $undefined-lit (convert-trail-from-W M) L \longleftrightarrow undefined-lit M L$   
**by** (auto simp: defined-lit-map)

**lemma** *lit-of-convert-marked-lit-from-NOT*[iff]:  
 $lit-of (convert-marked-lit-from-NOT L) = lit-of L$   
**by** (cases L) auto

**sublocale**  $state_W \subseteq dpll-state$  *convert-trail-from-W*  $\circ$  *trail clauses*  
 $\lambda L S. cons-trail (convert-marked-lit-from-NOT L) S$   
 $\lambda S. tl-trail S$   
 $\lambda C S. add-learned-cls C S$   
 $\lambda C S. remove-cls C S$   
**by** *unfold-locales auto*

**sublocale**  $cdcl_W-ops \subseteq cdcl_{NOT}-merge-bj-learn-ops$  *convert-trail-from-W*  $\circ$  *trail clauses*  
 $\lambda L S. cons-trail (convert-marked-lit-from-NOT L) S$   
 $\lambda S. tl-trail S$   
 $\lambda C S. add-learned-cls C S$   
 $\lambda C S. remove-cls C S$   
 $\lambda -. True$   
 $\lambda S. conflicting S = C-True \ \lambda C L S. backjump-l-cond C L S$   
 $\wedge distinct-mset (C + \{\#L\}) \wedge \neg tautology (C + \{\#L\})$   
**by** *unfold-locales*

**sublocale**  $cdcl_W-ops \subseteq cdcl_{NOT}-merge-bj-learn-proxy$  *convert-trail-from-W*  $\circ$  *trail clauses*  
 $\lambda L S. cons-trail (convert-marked-lit-from-NOT L) S$   
 $\lambda S. tl-trail S$   
 $\lambda C S. add-learned-cls C S$   
 $\lambda C S. remove-cls C S$   
 $\lambda -. True$   
 $\lambda S. conflicting S = C-True \ backjump-l-cond inv_{NOT}$

**proof** (*unfold-locales, goal-cases*)

**case** 2

**then show** ?case **using**  $cdcl_{NOT}-merged-bj-learn-no-dup-inv$  **by** auto

**next**

**case** (1  $C' S C F' K F L$ )

**moreover**

**let** ? $C' = remdups-mset C'$

**have**  $L \notin \# C'$

**using**  $\langle F \models_{as} CNot C' \rangle \langle undefined-lit F L \rangle$  *Marked-Propagated-in-iff-in-lits-of*  
 $in-CNot-implies-uminus(2)$  **by** blast

**then have**  $distinct-mset (?C' + \{\#L\})$

**by** (metis count-mset-set(3) distinct-mset-remdups-mset distinct-mset-single-add  
less-irrefl-nat mem-set-mset-iff remdups-mset-def)

**moreover**

**have** *no-dup F*

**using**  $\langle inv_{NOT} S \rangle \langle (convert-trail-from-W \circ trail) S = F' @ Marked K () \# F \rangle$

**unfolding**  $inv_{NOT}-def$

**by** (smt comp-apply distinct.simps(2) distinct-append list.simps(9) map-append  
no-dup-convert-from-W)

**then have** *consistent-interp (lits-of F)*

**using** *distinctconsistent-interp* **by** blast

**then have**  $\neg tautology (C')$

```

using  $\langle F \models_{as} CNot\ C' \rangle$  consistent-CNot-not-tautology true-annots-true-cls by blast
then have  $\neg$  tautology  $(?C' + \{\#L\# \})$ 
using  $\langle F \models_{as} CNot\ C' \rangle$  undefined-lit F L by (metis CNot-remdups-mset
Marked-Propagated-in-iff-in-lits-of add.commute in-CNot-uminus tautology-add-single
tautology-remdups-mset true-annot-singleton true-annots-def)
show ?case
proof –
  have f2: no-dup  $((convert-trail-from-W \circ trail)\ S)$ 
    using  $\langle inv_{NOT}\ S \rangle$  unfolding inv_{NOT}-def by simp
  have f3: atm-of  $L \in$  atms-of-mu  $(clauses\ S)$ 
     $\cup$  atm-of ‘lits-of  $((convert-trail-from-W \circ trail)\ S)$ ’
    using  $\langle (convert-trail-from-W \circ trail)\ S = F' @ Marked\ K\ () \# F \rangle$ 
     $\langle atm-of\ L \in atms-of-mu\ (clauses\ S) \cup atm-of\ 'lits-of\ (F' @ Marked\ K\ () \# F) \rangle$  by presburger
  have f4: clauses  $S \models_{pm}$  remdups-mset  $C' + \{\#L\# \}$ 
    by (metis (no-types)  $\langle L \notin \# C' \rangle \langle clauses\ S \models_{pm}\ C' + \{\#L\# \} \rangle$  remdups-mset-singleton-sum(2)
    true-clss-cls-remdups-mset union-commute)
  have  $F \models_{as}\ CNot\ (remdups-mset\ C')$ 
    by (simp add:  $\langle F \models_{as}\ CNot\ C' \rangle$ )
  then show ?thesis
    using f4 f3 f2  $\neg$  tautology  $(remdups-mset\ C' + \{\#L\# \})$  backjump-l.intros calculation(2–5,9)
    state-eq_{NOT}-ref by blast
qed
qed

```

```

sublocale cdclW-ops  $\subseteq$  cdclNOT-merge-bj-learn-proxy2 convert-trail-from-W o trail clauses
   $\lambda L\ S.$  cons-trail  $(convert-marked-lit-from-NOT\ L)\ S$ 
   $\lambda S.$  tl-trail  $S$ 
   $\lambda C\ S.$  add-learned-cls  $C\ S$ 
   $\lambda C\ S.$  remove-cls  $C\ S\ \lambda -.$  True invNOT
   $\lambda -\ S.$  conflicting  $S = C-True\ backjump-l-cond$ 
by unfold-locales

```

```

sublocale cdclW-ops  $\subseteq$  cdclNOT-merge-bj-learn convert-trail-from-W o trail clauses
   $\lambda L\ S.$  cons-trail  $(convert-marked-lit-from-NOT\ L)\ S$ 
   $\lambda S.$  tl-trail  $S$ 
   $\lambda C\ S.$  add-learned-cls  $C\ S$ 
   $\lambda C\ S.$  remove-cls  $C\ S\ \lambda -.$  True invNOT
   $\lambda -\ S.$  conflicting  $S = C-True\ backjump-l-cond$ 
apply unfold-locales
  using dp11-bj-no-dup apply simp
using cdclNOT.simps cdclNOT-no-dup by auto

```

```

context cdclW-ops
begin

```

Notations are lost while proving locale inclusion:

```

notation state-eqNOT (infix  $\sim_{NOT}$  50)

```

## 19.2 More lemmas conflict-propagate and backjumping

### 19.2.1 Termination

```

lemma cdclW-cp-normalized-element-all-inv:
  assumes inv: cdclW-all-struct-inv  $S$ 
  obtains  $T$  where full cdclW-cp  $S\ T$ 
  using assms cdclW-cp-normalized-element unfolding cdclW-all-struct-inv-def by blast

```

**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 = C-True then 0 else 1)*  
*> length (trail T) + (if conflicting T = C-True then 0 else 1)*  
**using** *assms* **by** (*induction rule: cdcl<sub>W</sub>-bj.induct*)  
*(fastforce dest:arg-cong[of - - length]*  
*intro: get-all-marked-decomposition-exists-prepend*  
*elim!: backtrack-levE*  
*simp: cdcl<sub>W</sub>-M-level-inv-def)+*

**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}*  
**apply** (*rule wfP-if-measure[of λ-. True*  
*- λT. length (trail T) + (if conflicting T = C-True then 0 else 1), simplified]*)  
**using** *cdcl<sub>W</sub>-bj-measure* **by** *blast*

**lemma** *cdcl<sub>W</sub>-bj-exists-normal-form*:

**assumes** *lev: cdcl<sub>W</sub>-M-level-inv S*  
**shows**  $\exists T. \text{full } \text{cdcl}_W\text{-bj } S \ T$

**proof** –

**obtain** *T* **where** *T: full (λa b. cdcl<sub>W</sub>-bj a b ∧ cdcl<sub>W</sub>-M-level-inv a) S T*  
**using** *wf-exists-normal-form-full[OF wf-cdcl<sub>W</sub>-bj]* **by** *auto*  
**then have** *cdcl<sub>W</sub>-bj\*\* S T*  
**by** (*auto dest: rtrancp-and-rtrancp-left simp: full-def*)

**moreover**

**then have** *cdcl<sub>W</sub>\*\* S T*  
**using** *mono-rtrancp[of cdcl<sub>W</sub>-bj cdcl<sub>W</sub>] cdcl<sub>W</sub>.simps* **by** *blast*  
**then have** *cdcl<sub>W</sub>-M-level-inv T*  
**using** *rtrancp-cdcl<sub>W</sub>-consistent-inv lev* **by** *auto*  
**ultimately show** *?thesis* **using** *T unfolding full-def* **by** *auto*

**qed**

**lemma** *rtrancp-skip-state-decomp*:

**assumes** *skip\*\* S T* **and** *no-dup (trail S)*  
**shows**  
 $\exists M. \text{trail } S = M @ \text{trail } T \wedge (\forall m \in \text{set } M. \neg \text{is-marked } m)$  **and**  
*T ~ delete-trail-and-rebuild (trail T) S*  
**using** *assms* **by** (*induction rule: rtrancp-induct*) (*auto simp del: state-simp simp: state-eq-def*)+

## 19.2.2 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*

**using** *assms*

**proof** *induction*

**case** *base*

**then show** *?case* **by** *simp*

**next**

**case** (*step T V*) **note** *st = this(1)* **and** *skip = this(2)* **and** *IH = this(3)* **and** *bt = this(4)* **and**  
*inv = this(5)*



```

have skip** S V
  using st skip by auto
then have cdclW-all-struct-inv V
  using rtrancp-mono[of skip cdclW] assms(3) rtrancp-cdclW-all-struct-inv-inv mono-rtrancp
  by (auto dest!: bj other cdclW-bj.skip)
then have cdclW-M-level-inv V
  unfolding cdclW-all-struct-inv-def by auto
then obtain N k M1 M2 K D L U i where
  V: state V = (trail V, N, U, k, C-Clause (D + {#L#})) and
  W: state W = (Propagated L (D + {#L#}) # M1, N, {#D + {#L#}#} + U,
    get-maximum-level D (trail V), C-True) and
  decomp: (Marked K (Suc i) # M1, M2)
    ∈ set (get-all-marked-decomposition (trail V)) and
  k = get-maximum-level (D + {#L#}) (trail V) and
  lev-L: get-level L (trail V) = k and
  undef: undefined-lit M1 L and
  W ~ cons-trail (Propagated L (D + {#L#}))
    (reduce-trail-to M1 (add-learned-cls (D + {#L#}))
      (update-backtrack-lvl (get-maximum-level D (trail V)) (update-conflicting C-True V))) and
  lev-l-D: backtrack-lvl V = get-maximum-level (D + {#L#}) (trail V) and
  conflicting V = C-Clause (D + {#L#}) and
  i: i = get-maximum-level D (trail V)
  using bt by (elim backtrack-levE) (auto simp: cdclW-M-level-inv-decomp)
let ?D = (D + {#L#})
obtain L' C' where
  T: state T = (Propagated L' C' # trail V, N, U, k, C-Clause ?D) and
  V ~ tl-trail T and
  -L' ∉ # ?D and
  ?D ≠ {#}
  using skip V by force

let ?M = Propagated L' C' # trail V
have cdclW** S T using bj cdclW-bj.skip mono-rtrancp[of skip cdclW S T] other st by meson
then have inv': cdclW-all-struct-inv T
  using rtrancp-cdclW-all-struct-inv-inv inv by blast
have M-lev: cdclW-M-level-inv T using inv' unfolding cdclW-all-struct-inv-def by auto
then have n-d': no-dup ?M
  using T unfolding cdclW-M-level-inv-def by auto

have k > 0
  using decomp M-lev T V unfolding cdclW-M-level-inv-def by auto
then have atm-of L ∈ atm-of ' lits-of (trail V)
  using lev-L get-rev-level-ge-0-atm-of-in V by fastforce
then have L-L': atm-of L ≠ atm-of L'
  using n-d' unfolding lits-of-def by auto
have L'-M: atm-of L' ∉ atm-of ' lits-of (trail V)
  using n-d' unfolding lits-of-def by auto
have ?M ⊨as CNot ?D
  using inv' T unfolding cdclW-conflicting-def cdclW-all-struct-inv-def by auto
then have L' ∉ # ?D
  using L-L' L'-M unfolding true-annots-def by (auto simp add: true-annot-def true-cls-def
    atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set Ball-mset-def
    split: split-if-asm)
have [simp]: trail (reduce-trail-to M1 T) = M1
  by (metis (mono-tags, lifting) One-nat-def Pair-inject T ⟨V ~ tl-trail T⟩ decomp

```

```

    diff-less in-get-all-marked-decomposition-trail-update-trail length-greater-0-conv
    length-tl lessI list.distinct(1) reduce-trail-to-length-ne state-eq-trail
    trail-reduce-trail-to-length-le trail-tl-trail)
  have skip** S V
    using st skip by auto
  have no-dup (trail S)
    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
  then have [simp]: init-cls S = N and [simp]: learned-cls S = U
    using rtrancp-skip-state-decomp[OF ⟨skip** S V⟩] V
    by (auto simp del: state-simp simp: state-eq-def)
  then have W-S: W ~ cons-trail (Propagated L (D + {#L#})) (reduce-trail-to M1
    (add-learned-cls (D + {#L#}) (update-backtrack-lvl i (update-conflicting C-True T))))
    using W i T undef M-lev by (auto simp del: state-simp simp: state-eq-def cdclW-M-level-inv-def)

  obtain M2' where
    (Marked K (i+1) # M1, M2') ∈ set (get-all-marked-decomposition ?M)
    using decomp V by (cases hd (get-all-marked-decomposition (trail V)),
      cases get-all-marked-decomposition (trail V)) auto
  moreover
    from L-L' have get-level L ?M = k
      using lev-L ⟨-L' ∉ # ?D⟩ V by (auto split: split-if-asm)
  moreover
    have atm-of L' ∉ atms-of D
      using ⟨L' ∉ # ?D⟩ ⟨-L' ∉ # ?D⟩ by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
        atms-of-def)
    then have get-level L ?M = get-maximum-level (D + {#L#}) ?M
      using lev-l-D[symmetric] L-L' V lev-L by simp
  moreover have i = get-maximum-level D ?M
    using i ⟨atm-of L' ∉ atms-of D⟩ by auto
  moreover

  ultimately have backtrack T W
    using T(1) W-S by blast
  then show ?thesis using IH inv by blast
qed

lemma fst-get-all-marked-decomposition-prepend-not-marked:
  assumes ∀ m ∈ set MS. ¬ is-marked m
  shows set (map fst (get-all-marked-decomposition M))
    = set (map fst (get-all-marked-decomposition (MS @ M)))
  using assms apply (induction MS rule: marked-lit-list-induct)
  apply auto[2]
  by (case-tac get-all-marked-decomposition (xs @ M)) simp-all

See also  $\llbracket \text{skip}^{**} ?S ?T; \text{backtrack} ?T ?W; \text{cdcl}_W\text{-all-struct-inv} ?S \rrbracket \implies \text{backtrack} ?S ?W$ 

lemma rtrancp-skip-backtrack-backtrack-end:
  assumes
    skip: skip** S T and
    bt: backtrack S W and
    inv: cdclW-all-struct-inv S
  shows backtrack T W
  using assms
proof -
  have M-lev: cdclW-M-level-inv S
    using bt inv unfolding cdclW-all-struct-inv-def by (auto elim!: backtrack-levE)

```

**then obtain**  $k M M1 M2 K i D L N U$  **where**  
*S*: state  $S = (M, N, U, k, C\text{-Clause } (D + \{\#L\#\}))$  **and**  
*W*: state  $W = (\text{Propagated } L \ (D + \{\#L\#\})) \# M1, N, \{\#D + \{\#L\#\}\# + U,$   
 $\text{get-maximum-level } D M, C\text{-True})$  **and**  
*decomp*:  $(\text{Marked } K \ (i+1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$  **and**  
*lev-l*:  $\text{get-level } L M = k$  **and**  
*lev-l-D*:  $\text{get-level } L M = \text{get-maximum-level } (D + \{\#L\#\}) M$  **and**  
*i*:  $i = \text{get-maximum-level } D M$  **and**  
*undef*:  $\text{undefined-lit } M1 L$   
**using** *bt* **by**  $(\text{elim backtrack-levE}) \ (\text{force simp: cdcl}_W\text{-M-level-inv-def}) +$   
**let**  $?D = (D + \{\#L\#\})$   
  
**have**  $[simp]: \text{no-dup } (\text{trail } S)$   
**using** *M-lev* **by**  $(\text{auto simp: cdcl}_W\text{-M-level-inv-decomp})$   
**have**  $\text{cdcl}_W\text{-all-struct-inv } T$   
**using** *mono-rtrancp* $[\text{of skip cdcl}_W]$  **by**  $(\text{smt bj cdcl}_W\text{-bj.skip inv local.skip other}$   
 $\text{rtrancp-cdcl}_W\text{-all-struct-inv-inv})$   
**then have**  $[simp]: \text{no-dup } (\text{trail } T)$   
**unfolding**  $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$  **by** *auto*  
  
**obtain**  $MS M_T$  **where**  $M: M = MS @ M_T$  **and**  $M_T: M_T = \text{trail } T$  **and**  $nm: \forall m \in \text{set } MS. \neg \text{is-marked}$   
 $m$   
**using** *rtrancp-skip-state-decomp* $(1)[\text{OF skip}] S M\text{-lev}$  **by** *auto*  
**have** *T*: state  $T = (M_T, N, U, k, C\text{-Clause } ?D)$   
**using**  $M_T \text{rtrancp-skip-state-decomp}(2)[\text{of } S T] \text{skip } S$   
**by**  $(\text{auto simp del: state-simp simp: state-eq-def})$   
  
**have**  $\text{cdcl}_W\text{-all-struct-inv } T$   
**apply**  $(\text{rule rtrancp-cdcl}_W\text{-all-struct-inv-inv}[\text{OF - inv}])$   
**using** *bj cdcl* $_W\text{-bj.skip local.skip other rtrancp-mono}[\text{of skip cdcl}_W]$  **by** *blast*  
**then have**  $M_T \models_{as} CNot ?D$   
**unfolding**  $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-conflicting-def}$  **using** *T* **by** *blast*  
**have**  $\forall L \in \#?D. \text{atm-of } L \in \text{atm-of ' lits-of } M_T$   
**proof** –  
**have**  $f1: \bigwedge l. \neg M_T \models_a \{\#- l\# \} \vee \text{atm-of } l \in \text{atm-of ' lits-of } M_T$   
**by**  $(\text{simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-lit-of-true-annot}$   
 $\text{lits-of-def})$   
**have**  $\bigwedge l. l \notin \# D \vee - l \in \text{lits-of } M_T$   
**using**  $\langle M_T \models_{as} CNot (D + \{\#L\#\}) \rangle \text{multi-member-split}$  **by** *fastforce*  
**then show** *?thesis*  
**using** *f1* **by**  $(\text{meson } \langle M_T \models_{as} CNot (D + \{\#L\#\}) \rangle \text{ball-msetI true-annots-CNot-all-atms-defined})$   
**qed**  
**moreover have** *no-dup* *M*  
**using** *inv S* **unfolding**  $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$  **by** *auto*  
**ultimately have**  $\forall L \in \#?D. \text{atm-of } L \notin \text{atm-of ' lits-of } MS$   
**unfolding** *M* **unfolding** *lits-of-def* **by** *auto*  
**then have**  $H: \bigwedge L. L \in \#?D \implies \text{get-level } L M = \text{get-level } L M_T$   
**unfolding** *M* **by**  $(\text{fastforce simp: lits-of-def})$   
**have**  $[simp]: \text{get-maximum-level } ?D M = \text{get-maximum-level } ?D M_T$   
**by**  $(\text{metis } \langle M_T \models_{as} CNot (D + \{\#L\#\}) \rangle M nm \text{ball-msetI true-annots-CNot-all-atms-defined}$   
 $\text{get-maximum-level-skip-un-marked-not-present})$   
  
**have** *lev-l'*:  $\text{get-level } L M_T = k$   
**using** *lev-l* **by**  $(\text{auto simp: } H)$   
**have**  $[simp]: \text{trail } (\text{reduce-trail-to } M1 T) = M1$

```

using  $T$  decomp  $M$   $nm$  by (smt  $M_T$  append-assoc beginning-not-marked-invert
  get-all-marked-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp)
have  $W$ :  $W \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \})) (\text{reduce-trail-to } M1$ 
  (add-learned-cls ( $D + \{\#L\# \}$ ) (update-backtrack-lvl  $i$  (update-conflicting  $C\text{-True } T$ ))))
using  $W$   $T$   $i$  decomp undef by (auto simp del: state-simp simp: state-eq-def)

have  $\text{lev-}l\text{-}D'$ : get-level  $L$   $M_T = \text{get-maximum-level } (D + \{\#L\# \}) M_T$ 
using  $\text{lev-}l\text{-}D$  by (auto simp:  $H$ )
have [simp]: get-maximum-level  $D$   $M = \text{get-maximum-level } D M_T$ 
proof –
  have  $\bigwedge ms \ m. \neg (ms::('v, \text{nat}, 'v \text{ literal multiset}) \text{ marked-lit list}) \models_{as} CNot \ m$ 
     $\vee (\forall l \in \#m. \text{atm-of } l \in \text{atm-of } ' \text{ lits-of } ms)$ 
  by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-CNot-implies-uminus(2))
  then have  $\forall l \in \#D. \text{atm-of } l \in \text{atm-of } ' \text{ lits-of } M_T$ 
    using  $\langle M_T \models_{as} CNot (D + \{\#L\# \}) \rangle$  by auto
  then show ?thesis
    by (metis  $M$  get-maximum-level-skip-un-marked-not-present  $nm$ )
  qed
then have  $i'$ :  $i = \text{get-maximum-level } D M_T$ 
using  $i$  by auto
have Marked  $K (i + 1) \# M1 \in \text{set } (\text{map } \text{fst } (\text{get-all-marked-decomposition } M))$ 
using Set.imageI[OF decomp, of fst] by auto
then have Marked  $K (i + 1) \# M1 \in \text{set } (\text{map } \text{fst } (\text{get-all-marked-decomposition } M_T))$ 
using fst-get-all-marked-decomposition-prepend-not-marked[OF nm] unfolding  $M$  by auto
then obtain  $M2'$  where decomp': (Marked  $K (i+1) \# M1, M2'$ )  $\in \text{set } (\text{get-all-marked-decomposition } M_T)$ 
by auto
then show backtrack  $T$   $W$ 
using backtrack.intros[OF T decomp' lev-l']  $\text{lev-}l\text{-}D' \ i' \ W$  by force
qed

lemma cdclW-bj-decomp-resolve-skip-and-bj:
  assumes cdclW-bj**  $S$   $T$  and inv: cdclW-M-level-inv  $S$ 
  shows (skip-or-resolve**  $S$   $T$ 
     $\vee (\exists U. \text{skip-or-resolve** } S \ U \wedge \text{backtrack } U \ T)$ )
  using assms
proof induction
  case base
  then show ?case by simp
next
  case (step  $T$   $U$ ) note  $st = \text{this}(1)$  and  $bj = \text{this}(2)$  and  $IH = \text{this}(3)$ 
  have  $IH$ : skip-or-resolve**  $S$   $T$ 
  proof –
    { assume ( $\exists U. \text{skip-or-resolve** } S \ U \wedge \text{backtrack } U \ T$ )
      then obtain  $V$  where
         $bt$ : backtrack  $V$   $T$  and
        skip-or-resolve**  $S$   $V$ 
      by blast
      have cdclW**  $S$   $V$ 
      using (skip-or-resolve**  $S$   $V$ ) rtranclp-skip-or-resolve-rtranclp-cdclW by blast
      then have cdclW-M-level-inv  $V$  and cdclW-M-level-inv  $S$ 
      using rtranclp-cdclW-consistent-inv inv by blast+
      with  $bj$   $bt$  have False using backtrack-no-cdclW-bj by simp
    }
  then show ?thesis using  $IH$  inv by blast

```

```

qed
show ?case
using bj
proof (cases rule: cdclW-bj.cases)
  case backtrack
  then show ?thesis using IH by blast
qed (metis (no-types, lifting) IH rtranclp.simps)+
qed

lemma resolve-skip-deterministic:
  resolve S T  $\implies$  skip S U  $\implies$  False
by fastforce

lemma backtrack-unique:
  assumes
    bt-T: backtrack S T and
    bt-U: backtrack S U and
    inv: cdclW-all-struct-inv S
  shows T  $\sim$  U
proof -
  have lev: cdclW-M-level-inv S
  using inv unfolding cdclW-all-struct-inv-def by auto
  then obtain M N U' k D L i K M1 M2 where
    S: state S = (M, N, U', k, C-Clause (D + {#L#})) and
    decomp: (Marked K (i+1) # M1, M2)  $\in$  set (get-all-marked-decomposition M) and
    get-level L M = k and
    get-level L M = get-maximum-level (D+{#L#}) M and
    get-maximum-level D M = i and
    T: state T = (Propagated L ((D+{#L#})) # M1, N, {#D + {#L#}#} + U', i, C-True) and
    undef: undefined-lit M1 L
  using bt-T by (elim backtrack-levE) (force simp: cdclW-M-level-inv-def)+

  obtain D' L' i' K' M1' M2' where
    S': state S = (M, N, U', k, C-Clause (D' + {#L'#})) and
    decomp': (Marked K' (i'+1) # M1', M2')  $\in$  set (get-all-marked-decomposition M) and
    get-level L' M = k and
    get-level L' M = get-maximum-level (D'+{#L'#}) M and
    get-maximum-level D' M = i' and
    U: state U = (Propagated L' ((D'+{#L'#})) # M1', N, {#D' + {#L'#}#} + U', i', C-True) and
    undef: undefined-lit M1' L'
  using bt-U lev S by (elim backtrack-levE) (force simp: cdclW-M-level-inv-def)+
  obtain c where M: M = c @ M2 @ Marked K (i + 1) # M1
  using decomp by auto
  obtain c' where M': M = c' @ M2' @ Marked K' (i' + 1) # M1'
  using decomp' by auto
  have marked: get-all-levels-of-marked M = rev [1..k]
  using inv S unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
  then have i < k
  unfolding M
  by (force simp add: rev-swap[symmetric] dest!: arg-cong[of - - set])

  have [simp]: L = L'
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then have L'  $\in$  # D

```

```

    using S unfolding S' by (fastforce simp: multiset-eq-iff split: split-if-asm)
  then have get-maximum-level D M  $\geq k$ 
    using  $\langle \text{get-level } L' M = k \rangle$  get-maximum-level-ge-get-level by blast
  then show False using  $\langle \text{get-maximum-level } D M = i \rangle \langle i < k \rangle$  by auto
qed
then have [simp]: D = D'
  using S S' by auto
have [simp]: i=i' using  $\langle \text{get-maximum-level } D' M = i' \rangle \langle \text{get-maximum-level } D M = i \rangle$  by auto

```

Automation in a step later...

```

have H:  $\bigwedge a A B. \text{insert } a A = B \implies a : B$ 
  by blast
have get-all-levels-of-marked (c@M2) = rev [i+2..1+k] and
  get-all-levels-of-marked (c'@M2') = rev [i+2..1+k]
  using marked unfolding M
  using marked unfolding M'
  unfolding rev-swap[symmetric] by (auto dest: append-cons-eq-upt-length-i-end)
from arg-cong[OF this(1), of set] arg-cong[OF this(2), of set]
have
  dropWhile ( $\lambda L. \neg \text{is-marked } L \vee \text{level-of } L \neq \text{Suc } i$ ) (c @ M2) = [] and
  dropWhile ( $\lambda L. \neg \text{is-marked } L \vee \text{level-of } L \neq \text{Suc } i$ ) (c' @ M2') = []
  unfolding dropWhile-eq-Nil-conv Ball-def
  by (intro allI; case-tac x; auto dest!: H simp add: in-set-conv-decomp)+

then have M1 = M1'
  using arg-cong[OF M, of dropWhile ( $\lambda L. \neg \text{is-marked } L \vee \text{level-of } L \neq \text{Suc } i$ )]
  unfolding M' by auto
then show ?thesis using T U by (auto simp del: state-simp simp: state-eq-def)
qed

```

lemma if-can-apply-backtrack-no-more-resolve:

```

assumes
  skip: skip** S U and
  bt: backtrack S T and
  inv: cdclW-all-struct-inv S
shows  $\neg \text{resolve } U V$ 
proof (rule ccontr)
  assume resolve:  $\neg \neg \text{resolve } U V$ 

```

obtain *L C M N U' k D* where

```

  U: state U = (Propagated L ( (C + {#L#}))) # M, N, U', k, C-Clause (D + {#-L#})) and
  get-maximum-level D (Propagated L ( (C + {#L#}))) # M) = k and
  state V = (M, N, U', k, C-Clause (D # $\cup$  C))
  using resolve by auto

```

have cdcl<sub>W</sub>-all-struct-inv *U*

```

  using mono-rtrancp[of skip cdclW] by (meson bj cdclW-bj.skip inv local.skip other
    rtrancp-cdclW-all-struct-inv-inv)

```

then have [iff]: no-dup (trail *S*) cdcl<sub>W</sub>-*M*-level-inv *S* and [iff]: no-dup (trail *U*)

```

  using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by blast+

```

then have

```

S: init-clss S = N
  learned-clss S = U'
  backtrack-lvl S = k
  conflicting S = C-Clause (D + {#-L#})

```

```

  using rtrancp-skip-state-decomp(2)[OF skip] U by (auto simp del: state-simp simp: state-eq-def)

```

**obtain**  $M_0$  **where**  
*tr-S*: trail  $S = M_0 @$  trail  $U$  **and**  
*nm*:  $\forall m \in \text{set } M_0. \neg \text{is-marked } m$   
**using** *rtranclp-skip-state-decomp*[*OF skip*] **by** *blast*

**obtain**  $M' D' L' i K M1 M2$  **where**  
*S'*: state  $S = (M', N, U', k, C\text{-Clause } (D' + \{\#L'\#\}))$  **and**  
*decomp*:  $(\text{Marked } K (i+1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M')$  **and**  
*get-level*  $L' M' = k$  **and**  
*get-level*  $L' M' = \text{get-maximum-level } (D' + \{\#L'\#\}) M'$  **and**  
*get-maximum-level*  $D' M' = i$  **and**  
*undef*: undefined-lit  $M1 L'$  **and**  
*T*: state  $T = (\text{Propagated } L' (D' + \{\#L'\#\}) \# M1, N, \{\#D' + \{\#L'\#\}\# + U', i, C\text{-True})$   
**using** *bt S* **by** (*force elim!*: *backtrack-levE*)

**obtain**  $c$  **where**  $M: M' = c @ M2 @ \text{Marked } K (i + 1) \# M1$   
**using** *get-all-marked-decomposition-exists-prepend*[*OF decomp*] **by** *auto*  
**have** *marked*: *get-all-levels-of-marked*  $M' = \text{rev } [1..<1+k]$   
**using** *inv S' unfolding cdcl<sub>W</sub>-all-struct-inv-def cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**then have**  $i < k$   
**unfolding**  $M$  **by** (*force simp add: rev-swap*[*symmetric*] *dest!*: *arg-cong*[*of - - set*])

**have**  $DD': D' + \{\#L'\#\} = D + \{\#-L\#\}$   
**using**  $S S'$  **by** *auto*  
**have** [*simp*]:  $L' = -L$   
**proof** (*rule ccontr*)  
**assume**  $\neg ?thesis$   
**then have**  $-L \in \# D'$   
**using**  $DD'$  **by** (*metis add-diff-cancel-right' diff-single-trivial diff-union-swap multi-self-add-other-not-self*)

**moreover**  
**have**  $M': M' = M_0 @ \text{Propagated } L ( (C + \{\#L\#\})) \# M$   
**using** *tr-S U S S'* **by** (*auto simp: lits-of-def*)  
**have** *no-dup*  $M'$   
**using** *inv U S' unfolding cdcl<sub>W</sub>-all-struct-inv-def cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**have** *atm-L-notin-M*: *atm-of*  $L \notin \text{atm-of } ( \text{lits-of } M )$   
**using**  $\langle \text{no-dup } M' \rangle M' U S S'$  **by** (*auto simp: lits-of-def*)  
**have** *get-all-levels-of-marked*  $M' = \text{rev } [1..<1+k]$   
**using** *inv U S' unfolding cdcl<sub>W</sub>-all-struct-inv-def cdcl<sub>W</sub>-M-level-inv-def* **by** *auto*  
**then have** *get-all-levels-of-marked*  $M = \text{rev } [1..<1+k]$   
**using** *nm M' S' U* **by** (*simp add: get-all-levels-of-marked-no-marked*)  
**then have** *get-lev-L*:  
*get-level*  $L (\text{Propagated } L ( (C + \{\#L\#\})) \# M) = k$   
**using** *get-level-get-rev-level-get-all-levels-of-marked*[*OF atm-L-notin-M, of [Propagated L ((C + {\#L\#}))]*] **by** *simp*  
**have** *atm-of*  $L \notin \text{atm-of } ( \text{lits-of } (\text{rev } M_0) )$   
**using**  $\langle \text{no-dup } M' \rangle M' U S'$  **by** (*auto simp: lits-of-def*)  
**then have** *get-level*  $L M' = k$   
**using** *get-rev-level-notin-end*[*of L rev M<sub>0</sub> 0*]  
 $\text{rev } M @ \text{Propagated } L ( (C + \{\#L\#\})) \# []$   
**using** *tr-S get-lev-L M' U S'* **by** (*simp add: nm lits-of-def*)

**ultimately have** *get-maximum-level*  $D' M' \geq k$   
**by** (*metis get-maximum-level-ge-get-level get-rev-level-uminus*)  
**then show** *False*  
**using**  $\langle i < k \rangle$  **unfolding**  $\langle \text{get-maximum-level } D' M' = i \rangle$  **by** *auto*

**qed**

```

have [simp]:  $D = D'$  using  $DD'$  by auto
have  $cdcl_W^{**} S U$ 
  using  $bj\ cdcl_W\text{-}bj.\text{skip}\ local.\text{skip}\ mono\text{-}rtranclp[of\ skip\ cdcl_W\ S\ U]$  other by meson
then have  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ U$ 
  using  $inv\ rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$  by blast
then have  $Propagated\ L\ (\ (C + \{\#L\#\}) \# M \models_{as} CNot\ (D' + \{\#L'\#\})$ 
  using  $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}conflicting\text{-}def\ U$  by auto
then have  $\forall L' \in \#D. atm\text{-}of\ L' \in atm\text{-}of\ ' \text{ lits-of } (Propagated\ L\ (\ (C + \{\#L\#\}) \# M)$ 
  by (metis  $CNot\text{-}plus\ CNot\text{-}singleton\ Un\text{-}insert\text{-}right\ \langle D = D' \rangle\ true\text{-}annots\text{-}insert\ ball\text{-}msetI$ 
     $atm\text{-}of\text{-}in\text{-}atm\text{-}of\text{-}set\text{-}iff\text{-}in\text{-}set\text{-}or\text{-}uminus\text{-}in\text{-}set\ in\text{-}CNot\text{-}implies\text{-}uminus(2)$ 
     $sup\text{-}bot.\text{comm}\text{-}neutral$ )
then have  $get\text{-}maximum\text{-}level\ D\ M' = k$ 
  using  $tr\text{-}S\ nm\ U\ S'$ 
     $get\text{-}maximum\text{-}level\text{-}skip\text{-}un\text{-}marked\text{-}not\text{-}present[of\ D$ 
       $Propagated\ L\ (\ (C + \{\#L\#\}) \# M\ M_0]$ 
  unfolding  $\langle get\text{-}maximum\text{-}level\ D\ (Propagated\ L\ (\ (C + \{\#L\#\}) \# M) = k \rangle$ 
  unfolding  $\langle D = D' \rangle$ 
  by simp
show False
  using  $\langle get\text{-}maximum\text{-}level\ D'\ M' = i \rangle \langle get\text{-}maximum\text{-}level\ D'\ M' = k \rangle \langle i < k \rangle$  by auto
qed

```

**lemma** *if-can-apply-resolve-no-more-backtrack:*

```

assumes
   $skip: skip^{**} S U$  and
   $resolve: resolve\ S\ T$  and
   $inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ 
shows  $\neg backtrack\ U\ V$ 
using  $assms$ 
by (meson  $if\text{-}can\text{-}apply\text{-}backtrack\text{-}no\text{-}more\text{-}resolve\ rtranclp.\text{rtrancl}\text{-}refl$ 
   $rtranclp\text{-}skip\text{-}backtrack\text{-}backtrack$ )

```

**lemma** *if-can-apply-backtrack-skip-or-resolve-is-skip:*

```

assumes
   $bt: backtrack\ S\ T$  and
   $skip: skip\text{-}or\text{-}resolve^{**} S U$  and
   $inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ 
shows  $skip^{**} S U$ 
using  $assms(2,3,1)$ 
by induction ( $simp\text{-}all\ add: if\text{-}can\text{-}apply\text{-}backtrack\text{-}no\text{-}more\text{-}resolve$ )

```

**lemma** *cdcl\_W-bj-bj-decomp:*

```

assumes  $cdcl_W\text{-}bj^{**} S W$  and  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ 
shows
   $(\exists T\ U\ V. (\lambda S\ T. skip\text{-}or\text{-}resolve\ S\ T \wedge no\text{-}step\ backtrack\ S)^{**} S\ T$ 
     $\wedge (\lambda T\ U. resolve\ T\ U \wedge no\text{-}step\ backtrack\ T)\ T\ U$ 
     $\wedge skip^{**} U\ V \wedge backtrack\ V\ W)$ 
   $\vee (\exists T\ U. (\lambda S\ T. skip\text{-}or\text{-}resolve\ S\ T \wedge no\text{-}step\ backtrack\ S)^{**} S\ T$ 
     $\wedge (\lambda T\ U. resolve\ T\ U \wedge no\text{-}step\ backtrack\ T)\ T\ U \wedge skip^{**} U\ W)$ 
   $\vee (\exists T. skip^{**} S\ T \wedge backtrack\ T\ W)$ 
   $\vee skip^{**} S\ W$  (is  $?RB\ S\ W \vee ?R\ S\ W \vee ?SB\ S\ W \vee ?S\ S\ W$ )
using  $assms$ 
proof induction
case base
then show ?case by simp

```



```

next
case (step W X) note st = this(1) and bj = this(2) and IH = this(3)[OF this(4)] and inv = this(4)

have  $\neg ?RB\ S\ W$  and  $\neg ?SB\ S\ W$ 
proof (clarify, goal-cases)
  case (1 T U V)
  have skip-or-resolve** S T
    using 1(1) by (auto dest!: rtrancpl-and-rtrancpl-left)
  then show False
    by (metis (no-types, lifting) 1(2) 1(4) 1(5) backtrack-no-cdclW-bj
      cdclW-all-struct-inv-def cdclW-all-struct-inv-inv cdclW-o.bj local.bj other
      resolve rtrancpl-cdclW-all-struct-inv-inv rtrancpl-skip-backtrack-backtrack
      rtrancpl-skip-or-resolve-rtrancpl-cdclW step.premis)
next
case 2
then show ?case by (meson assms(2) cdclW-all-struct-inv-def backtrack-no-cdclW-bj
  local.bj rtrancpl-skip-backtrack-backtrack)
qed
then have IH:  $?R\ S\ W \vee ?S\ S\ W$  using IH by blast

have cdclW** S W by (metis cdclW-o.bj mono-rtrancpl other st)
then have inv-W: cdclW-all-struct-inv W by (simp add: rtrancpl-cdclW-all-struct-inv-inv
  step.premis)
consider
  (BT) X' where backtrack W X'
| (skip) no-step backtrack W and skip W X
| (resolve) no-step backtrack W and resolve W X
using bj cdclW-bj.cases by meson
then show ?case
proof cases
  case (BT X')
  then consider
    (bt) backtrack W X
  | (sk) skip W X
  using bj if-can-apply-backtrack-no-more-resolve[of W W X' X] inv-W cdclW-bj.cases by fast
then show ?thesis
proof cases
  case bt
  then show ?thesis using IH by auto
next
  case sk
  then show ?thesis using IH by (meson rtrancpl-trans r-into-rtrancpl)
qed
next
case skip
then show ?thesis using IH by (meson rtrancpl.rtrancpl-into-rtrancpl)
next
case resolve note no-bt = this(1) and res = this(2)
consider
  (RS) T U where
    ( $\lambda S\ T.$  skip-or-resolve S T  $\wedge$  no-step backtrack S)** S T and
    resolve T U and
    no-step backtrack T and
    skip** U W
  | (S) skip** S W

```

```

using IH by auto
then show ?thesis
proof cases
  case (RS T U)
  have cdclW** S T
  using RS(1) cdclW-bj.resolve cdclW-o.bj other skip
  mono-rtrancpl[of (λS T. skip-or-resolve S T ∧ no-step backtrack S) cdclW S T]
  by meson
  then have cdclW-all-struct-inv U
  by (meson RS(2) cdclW-all-struct-inv-inv cdclW-bj.resolve cdclW-o.bj other
    rtrancpl-cdclW-all-struct-inv-inv step.prem)
  { fix U'
    assume skip** U U' and skip** U' W
    have cdclW-all-struct-inv U'
    using ⟨cdclW-all-struct-inv U⟩ ⟨skip** U U'⟩ rtrancpl-cdclW-all-struct-inv-inv
      cdclW-o.bj rtrancpl-mono[of skip cdclW] other skip by blast
    then have no-step backtrack U'
    using if-can-apply-backtrack-no-more-resolve[OF ⟨skip** U' W⟩] res by blast
  }
  with ⟨skip** U W⟩
  have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** U W
  proof induction
    case base
    then show ?case by simp
  next
    case (step V W) note st = this(1) and skip = this(2) and IH = this(3) and H = this(4)
    have ∧ U'. skip** U' V ⇒ skip** U' W
    using skip by auto
    then have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** U V
    using IH H by blast
    moreover have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** V W
    by (simp add: local.skip r-into-rtrancpl st step.prem)
    ultimately show ?case by simp
  qed
then show ?thesis
proof -
  have f1: ∀ p pa pb pc. ¬ p (pa) pb ∨ ¬ p** pb pc ∨ p** pa pc
  by (meson converse-rtrancpl-into-rtrancpl)
  have skip-or-resolve T U ∧ no-step backtrack T
  using RS(2) RS(3) by force
  then have (λp pa. skip-or-resolve p pa ∧ no-step backtrack p)** T W
  proof -
    have (∃ vr19 vr16 vr17 vr18. vr19 (vr16::'st) vr17 ∧ vr19** vr17 vr18
      ∧ ¬ vr19** vr16 vr18)
      ∨ ¬ (skip-or-resolve T U ∧ no-step backtrack T)
      ∨ ¬ (λuu uua. skip-or-resolve uu uua ∧ no-step backtrack uu)** U W
      ∨ (λuu uua. skip-or-resolve uu uua ∧ no-step backtrack uu)** T W
    by force
    then show ?thesis
    by (metis (no-types) ⟨(λS T. skip-or-resolve S T ∧ no-step backtrack S)** U W⟩
      ⟨skip-or-resolve T U ∧ no-step backtrack T⟩ f1)
  qed
  then have (λp pa. skip-or-resolve p pa ∧ no-step backtrack p)** S W
  using RS(1) by force

```

```

    then show ?thesis
      using no-bt res by blast
  qed
next
case S
{ fix U'
  assume skip** S U' and skip** U' W
  then have cdclW** S U'
    using mono-rtrancpl[of skip cdclW S U'] by (simp add: cdclW-o.bj other skip)
  then have cdclW-all-struct-inv U'
    by (metis (no-types, hide-lams) ⟨cdclW-all-struct-inv S⟩ rtrancpl-cdclW-all-struct-inv-inv)
  then have no-step backtrack U'
    using if-can-apply-backtrack-no-more-resolve[OF ⟨skip** U' W⟩] res by blast
}
with S
have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** S W
  proof induction
    case base
    then show ?case by simp
  next
  case (step V W) note st = this(1) and skip = this(2) and IH = this(3) and H = this(4)
  have ∧U'. skip** U' V ⇒ skip** U' W
    using skip by auto
  then have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** S V
    using IH H by blast
  moreover have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** V W
    by (simp add: local.skip r-into-rtrancpl st step.prem)
  ultimately show ?case by simp
  qed
  then show ?thesis using res no-bt by blast
qed
qed
qed

```

**Backjumping is confluent** lemma *cdcl<sub>W</sub>-bj-state-eq-compatible*:

```

assumes
  cdclW-bj S T and cdclW-M-level-inv S
  S ~ S' and
  T ~ T'
shows cdclW-bj S' T'
using assms
by induction (auto
  intro: skip-state-eq-compatible backtrack-state-eq-compatible resolve-state-eq-compatible)

```

**lemma** *trancpl-cdcl<sub>W</sub>-bj-state-eq-compatible*:

```

assumes
  cdclW-bj++ S T and inv: cdclW-M-level-inv S and
  S ~ S' and
  T ~ T'
shows cdclW-bj++ S' T'
using assms
proof (induction arbitrary: S' T')
  case base
  then show ?case

```

```

    using cdclW-bj-state-eq-compatible by blast
next
case (step T U) note IH = this(3)[OF this(4-5)]
have cdclW++ S T
  using trancpl-mono[of cdclW-bj cdclW] other step.hyps(1) by blast
then have cdclW-M-level-inv T
  using inv trancpl-cdclW-consistent-inv by blast
then have cdclW-bj++ T T'
  using  $\langle U \sim T' \rangle$  cdclW-bj-state-eq-compatible[of T U]  $\langle cdcl_W-bj T U \rangle$  by auto
then show ?case
  using IH[of T] by auto
qed

```

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
  cdclW-bj** S V and
  cdclW-bj** S T and
  n-s: no-step cdclW-bj V and
  inv: cdclW-all-struct-inv S
shows  $T \sim V \vee cdcl_W-bj^{**} T V$ 
using assms(2)
proof induction
case base
then show ?case by (simp add: assms(1))
next
case (step T U) note st = this(1) and s-o-r = this(2) and IH = this(3)
have cdclW** S T
  using st mono-rtrancpl[of cdclW-bj cdclW] other by blast
then have lev-T: cdclW-M-level-inv T
  using inv rtrancpl-cdclW-consistent-inv[of S T]
  unfolding cdclW-all-struct-inv-def by auto

```

**consider**

```

  (TV)  $T \sim V$ 
  | (bj-TV) cdclW-bj** T V
  using IH by blast
then show ?case
proof cases
case TV
have no-step cdclW-bj T
  using  $\langle cdcl_W-M-level-inv T \rangle$  n-s cdclW-bj-state-eq-compatible[of T - V] TV by auto
then show ?thesis
  using s-o-r by auto
next
case bj-TV
then obtain U' where
  T-U': cdclW-bj T U' and
  cdclW-bj** U' V
  using IH n-s s-o-r by (metis rtrancpl-unfold trancplD)
have cdclW** S T
  by (metis (no-types, hide-lams) bj mono-rtrancpl[of cdclW-bj cdclW] other st)
then have inv-T: cdclW-all-struct-inv T
  by (metis (no-types, hide-lams) inv rtrancpl-cdclW-all-struct-inv-inv)

```

```

have lev-U: cdclW-M-level-inv U
  using s-o-r cdclW-consistent-inv lev-T other by blast
show ?thesis
  using s-o-r
  proof cases
    case backtrack
      then obtain V0 where skip** T V0 and backtrack V0 V
        using IH if-can-apply-backtrack-skip-or-resolve-is-skip[OF backtrack - inv-T]
          cdclW-bj-decomp-resolve-skip-and-bj
          by (meson bj-TV cdclW-bj.backtrack inv-T lev-T n-s
            rtrancpl-skip-backtrack-backtrack-end)
      then have cdclW-bj** T V0 and cdclW-bj V0 V
        using rtrancpl-mono[of skip cdclW-bj] by blast+
      then show ?thesis
        using ⟨backtrack V0 V⟩ ⟨skip** T V0⟩ backtrack-unique inv-T local.backtrack
          rtrancpl-skip-backtrack-backtrack by auto
    next
      case resolve
        then have U ~ U'
          by (meson T-U' cdclW-bj.simps if-can-apply-backtrack-no-more-resolve inv-T
            resolve-skip-deterministic resolve-unique rtrancpl.rtrancpl-refl)
        then show ?thesis
          using ⟨cdclW-bj** U' V⟩ unfolding rtrancpl-unfold
          by (meson T-U' bj cdclW-consistent-inv lev-T other state-eq-ref state-eq-sym
            trancpl-cdclW-bj-state-eq-compatible)
    next
      case skip
        consider
          (sk) skip T U'
          | (bt) backtrack T U'
        using T-U' by (meson cdclW-bj.cases local.skip resolve-skip-deterministic)
        then show ?thesis
          proof cases
            case sk
              then show ?thesis
                using ⟨cdclW-bj** U' V⟩ unfolding rtrancpl-unfold
                by (meson T-U' bj cdclW-all-inv(3) cdclW-all-struct-inv-def inv-T local.skip other
                  trancpl-cdclW-bj-state-eq-compatible skip-unique state-eq-ref)
            next
              case bt
                have skip++ T U
                  using local.skip by blast
                then show ?thesis
                  using bt by (metis ⟨cdclW-bj** U' V⟩ backtrack inv-T trancpl-unfold-begin
                    rtrancpl-skip-backtrack-backtrack-end trancpl-into-rtrancpl)
          qed
        qed
      qed
    qed
  qed

```

**lemma** *cdcl<sub>W</sub>-bj-unique-normal-form*:  
**assumes**  
*ST*: *cdcl<sub>W</sub>-bj\*\* S T* and *SU*: *cdcl<sub>W</sub>-bj\*\* S U* and  
*n-s-U*: *no-step cdcl<sub>W</sub>-bj U* and

$n-s-T$ : *no-step*  $cdcl_W$ -bj  $T$  **and**  
 $inv$ :  $cdcl_W$ -all-struct- $inv$   $S$   
**shows**  $T \sim U$   
**proof** –  
**have**  $T \sim U \vee cdcl_W$ -bj\*\*  $T U$   
**using**  $ST SU cdcl_W$ -bj-strongly-confluent  $inv$   $n-s-U$  **by** *blast*  
**then show** *?thesis*  
**by** (*metis* (*no-types*)  $n-s-T$  *rtranclp-unfold state-eq-ref tranclp-unfold-begin*)  
**qed**

**lemma** *full-cdcl<sub>W</sub>-bj-unique-normal-form*:  
**assumes** *full*  $cdcl_W$ -bj  $S T$  **and** *full*  $cdcl_W$ -bj  $S U$  **and**  
 $inv$ :  $cdcl_W$ -all-struct- $inv$   $S$   
**shows**  $T \sim U$   
**using** *cdcl<sub>W</sub>-bj-unique-normal-form* *assms* **unfolding** *full-def* **by** *blast*

### 19.3 CDCL FW

**inductive**  $cdcl_W$ -merge-restart ::  $'st \Rightarrow 'st \Rightarrow bool$  **where**  
*fw-r-propagate*:  $propagate S S' \Longrightarrow cdcl_W$ -merge-restart  $S S' \mid$   
*fw-r-conflict*:  $conflict S T \Longrightarrow full\ cdcl_W$ -bj  $T U \Longrightarrow cdcl_W$ -merge-restart  $S U \mid$   
*fw-r-decide*:  $decide S S' \Longrightarrow cdcl_W$ -merge-restart  $S S' \mid$   
*fw-r-rf*:  $cdcl_W$ -rf  $S S' \Longrightarrow cdcl_W$ -merge-restart  $S S'$

**lemma** *cdcl<sub>W</sub>-merge-restart-cdcl<sub>W</sub>*:  
**assumes**  $cdcl_W$ -merge-restart  $S T$   
**shows**  $cdcl_W$ \*\*  $S T$   
**using** *assms*  
**proof** *induction*  
**case** (*fw-r-conflict*  $S T U$ ) **note**  $confl = this(1)$  **and**  $bj = this(2)$   
**have**  $cdcl_W S T$  **using**  $confl$  **by** (*simp* *add*:  $cdcl_W.intros$  *r-into-rtranclp*)  
**moreover**  
**have**  $cdcl_W$ -bj\*\*  $T U$  **using**  $bj$  **unfolding** *full-def* **by** *auto*  
**then have**  $cdcl_W$ \*\*  $T U$  **by** (*metis*  $cdcl_W$ -o.bj *mono-rtranclp* *other*)  
**ultimately show** *?case* **by** *auto*  
**qed** (*simp-all* *add*:  $cdcl_W$ -o.intros  $cdcl_W.intros$  *r-into-rtranclp*)

**lemma** *cdcl<sub>W</sub>-merge-restart-conflicting-true-or-no-step*:  
**assumes**  $cdcl_W$ -merge-restart  $S T$   
**shows**  $conflicting T = C-True \vee no\text{-}step\ cdcl_W T$   
**using** *assms*  
**proof** *induction*  
**case** (*fw-r-conflict*  $S T U$ ) **note**  $confl = this(1)$  **and**  $n-s = this(2)$   
**{** **fix**  $D V$   
**assume**  $cdcl_W U V$  **and**  $conflicting U = C-Clause D$   
**then have** *False*  
**using**  $n-s$  **unfolding** *full-def*  
**by** (*induction* *rule*:  $cdcl_W$ -all-rules-induct) (*auto* *dest*!:  $cdcl_W$ -bj.intros )  
**}**  
**then show** *?case* **by** (*cases*  $conflicting U$ ) *fastforce*+  
**qed** (*auto* *simp* *add*:  $cdcl_W$ -rf.simps)

**inductive**  $cdcl_W$ -merge ::  $'st \Rightarrow 'st \Rightarrow bool$  **where**  
*fw-propagate*:  $propagate S S' \Longrightarrow cdcl_W$ -merge  $S S' \mid$   
*fw-conflict*:  $conflict S T \Longrightarrow full\ cdcl_W$ -bj  $T U \Longrightarrow cdcl_W$ -merge  $S U \mid$   
*fw-decide*:  $decide S S' \Longrightarrow cdcl_W$ -merge  $S S' \mid$

*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*:

$\text{cdcl}_W\text{-merge } S \ T \implies \text{cdcl}_W\text{-merge-restart } S \ T$

**by** (*meson cdcl<sub>W</sub>-merge.cases cdcl<sub>W</sub>-merge-restart.simps forget*)

**lemma** *rtrancpl-cdcl<sub>W</sub>-merge-rtrancpl-cdcl<sub>W</sub>-merge-restart*:

$\text{cdcl}_W\text{-merge}^{**} \ S \ T \implies \text{cdcl}_W\text{-merge-restart}^{**} \ S \ T$

**using** *rtrancpl-mono*[*of cdcl<sub>W</sub>-merge cdcl<sub>W</sub>-merge-restart*] *cdcl<sub>W</sub>-merge-cdcl<sub>W</sub>-merge-restart* **by** *blast*

**lemma** *cdcl<sub>W</sub>-merge-rtrancpl-cdcl<sub>W</sub>*:

$\text{cdcl}_W\text{-merge } S \ T \implies \text{cdcl}_W^{**} \ S \ T$

**using** *cdcl<sub>W</sub>-merge-cdcl<sub>W</sub>-merge-restart cdcl<sub>W</sub>-merge-restart-cdcl<sub>W</sub>* **by** *blast*

**lemma** *rtrancpl-cdcl<sub>W</sub>-merge-rtrancpl-cdcl<sub>W</sub>*:

$\text{cdcl}_W\text{-merge}^{**} \ S \ T \implies \text{cdcl}_W^{**} \ S \ T$

**using** *rtrancpl-mono*[*of cdcl<sub>W</sub>-merge cdcl<sub>W</sub>^{\*\*}*] *cdcl<sub>W</sub>-merge-rtrancpl-cdcl<sub>W</sub>* **by** *auto*

**lemmas** *trail-reduce-trail-to<sub>NOT</sub>-add-cl<sub>s</sub><sub>NOT</sub>-unfolded*[*simp*] =

*trail-reduce-trail-to<sub>NOT</sub>-add-cl<sub>s</sub><sub>NOT</sub>*[*unfolded o-def*]

**lemma** *trail<sub>W</sub>-eq-reduce-trail-to<sub>NOT</sub>-eq*:

$\text{trail } S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ F \ S) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ F \ T)$

**proof** (*induction F S arbitrary: T rule: reduce-trail-to<sub>NOT</sub>.induct*)

**case** (*1 F S T*) **note** *IH = this(1)* **and** *tr = this(2)*

**then have**  $\square = \text{convert-trail-from-}W \ (\text{trail } S)$

$\vee \text{length } F = \text{length } (\text{convert-trail-from-}W \ (\text{trail } S))$

$\vee \text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ F \ (\text{tl-trail } S)) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ F \ (\text{tl-trail } T))$

**using** *IH* **by** (*metis (no-types) comp-apply trail-tl-trail*)

**then show**  $\text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ F \ S) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ F \ T)$

**using** *tr* **by** (*metis (no-types) comp-apply reduce-trail-to<sub>NOT</sub>.elims*)

**qed**

**lemma** *trail-reduce-trail-to<sub>NOT</sub>-add-learned-cl<sub>s</sub>*[*simp*]:

*no-dup* (*trail S*)  $\implies$

$\text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ M \ (\text{add-learned-cl}_S \ D \ S)) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} \ M \ S)$

**by** (*rule trail<sub>W</sub>-eq-reduce-trail-to<sub>NOT</sub>-eq simp*)

**lemma** *reduce-trail-to<sub>NOT</sub>-reduce-trail-convert*:

$\text{reduce-trail-to}_{\text{NOT}} \ C \ S = \text{reduce-trail-to } (\text{convert-trail-from-NOT } C) \ S$

**apply** (*induction C S rule: reduce-trail-to<sub>NOT</sub>.induct*)

**apply** (*subst reduce-trail-to<sub>NOT</sub>.simps, subst reduce-trail-to.simps*)

**by** (*auto simp: comp-def*)

**lemma** *reduce-trail-to-length*:

$\text{length } M = \text{length } M' \implies \text{reduce-trail-to } M \ S = \text{reduce-trail-to } M' \ S$

**apply** (*induction M S arbitrary: rule: reduce-trail-to.induct*)

**apply** (*case-tac trail S  $\neq \square$ ; case-tac length (trail S)  $\neq \text{length } M'$ ; simp*)

**by** (*simp-all add: reduce-trail-to-length-ne*)

**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**  $cdcl_{NOT}$ -merged-bj-learn  $S T$   
 $\vee$  (no-step  $cdcl_W$ -merge  $T \wedge$  conflicting  $T \neq C$ -True)  
**using**  $cdcl_W$  inv  
**proof** induction  
**case** (fw-propagate  $S T$ ) **note**  $propa = this(1)$   
**then obtain**  $M N U k L C$  **where**  
 $H$ : state  $S = (M, N, U, k, C$ -True) **and**  
 $CL$ :  $C + \{\#L\# \} \in \#$  clauses  $S$  **and**  
 $M$ - $C$ :  $M \models_{as} C$ Not  $C$  **and**  
 $undef$ : undefined-lit (trail  $S$ )  $L$  **and**  
 $T$ :  $T \sim$  cons-trail (Propagated  $L (C + \{\#L\# \})$ )  $S$   
**using**  $propa$  **by** auto  
**have**  $propagate_{NOT} S T$   
**apply** (rule  $propagate_{NOT}.propagate_{NOT}[of - C L]$ )  
**using**  $H CL T undef M$ - $C$  **by** (auto simp: state-eq $_{NOT}$ -def state-eq-def clauses-def  
simp del: state-simp $_{NOT}$  state-simp)  
**then show** ?case  
**using**  $cdcl_{NOT}$ -merged-bj-learn.intros(2) **by** blast  
**next**  
**case** (fw-decide  $S T$ ) **note**  $dec = this(1)$  **and**  $inv = this(2)$   
**then obtain**  $L$  **where**  
 $undef$ - $L$ : undefined-lit (trail  $S$ )  $L$  **and**  
 $atm$ - $L$ :  $atm$ -of  $L \in$   $atms$ -of- $\mu$  (init-clss  $S$ ) **and**  
 $T$ :  $T \sim$  cons-trail (Marked  $L (Suc$  (backtrack-lvl  $S$ )))  
(update-backtrack-lvl (Suc (backtrack-lvl  $S$ ))  $S$ )  
**by** auto  
**have**  $decide_{NOT} S T$   
**apply** (rule  $decide_{NOT}.decide_{NOT}$ )  
**using**  $undef$ - $L$  **apply** simp  
**using**  $atm$ - $L$   $inv$  **unfolding**  $cdcl_W$ -all-struct-inv-def no-strange-atm-def clauses-def **apply** auto[]  
**using**  $T undef$ - $L$  **unfolding** state-eq-def state-eq $_{NOT}$ -def **by** (auto simp: clauses-def)  
**then show** ?case **using**  $cdcl_{NOT}$ -merged-bj-learn-decide $_{NOT}$  **by** blast  
**next**  
**case** (fw-forget  $S T$ ) **note**  $rf = this(1)$  **and**  $inv = this(2)$   
**then obtain**  $M N C U k$  **where**  
 $S$ : state  $S = (M, N, \{\#C\# \} + U, k, C$ -True) **and**  
 $\neg M \models_{asm}$  clauses  $S$  **and**  
 $C \notin$  set (get-all-mark-of-propagated (trail  $S$ )) **and**  
 $C$ -init:  $C \notin \#$  init-clss  $S$  **and**  
 $C$ -le:  $C \in \#$  learned-clss  $S$  **and**  
 $T$ :  $T \sim$  remove-cls  $C S$   
**by** auto  
**have**  $init$ -clss  $S \models_{pm} C$   
**using**  $inv C$ -le **unfolding**  $cdcl_W$ -all-struct-inv-def  $cdcl_W$ -learned-clause-def  
**by** (meson mem-set-mset-iff true-clss-clss-in-imp-true-clss-clss)  
**then have**  $S$ - $C$ : clauses  $S -$  replicate-mset (count (clauses  $S$ )  $C$ )  $C \models_{pm} C$   
**using**  $C$ -init  $C$ -le **unfolding** clauses-def **by** (simp add: Un-Diff)  
**moreover have**  $H$ :  $init$ -clss  $S +$  (learned-clss  $S -$  replicate-mset (count (learned-clss  $S$ )  $C$ )  $C$ )  
 $=$   $init$ -clss  $S +$  learned-clss  $S -$  replicate-mset (count (learned-clss  $S$ )  $C$ )  $C$   
**using**  $C$ -le  $C$ -init **by** (metis clauses-def clauses-remove-clss diff-zero gr0I  
init-clss-remove-clss learned-clss-remove-clss plus-multiset.rep-eq replicate-mset-0  
semiring-normalization-rules(5))  
**have** forget $_{NOT} S T$   
**apply** (rule forget $_{NOT}.forget_{NOT}$ )  
**using**  $S$ - $C$  **apply** blast



```

    using  $S$  apply simp
    using  $\langle C \in \# \text{ learned-clss } S \rangle$  apply (simp add: clauses-def)
    using  $T$  C-le C-init by (auto
      simp: state-eq-def Un-Diff state-eqNOT-def clauses-def ac-simps  $H$ 
      simp del: state-simp state-simpNOT)
    then show ?case using cdclNOT-merged-bj-learn-forgetNOT by blast
next
case (fw-conflict  $S$   $T$   $U$ ) note confl = this(1) and bj = this(2) and inv = this(3)
obtain  $C_S$  where
  confl- $T$ : conflicting  $T = C\text{-Clause } C_S$  and
   $C_S$ :  $C_S \in \# \text{ clauses } S$  and
  tr- $S$ - $C_S$ : trail  $S \models_{as} C\text{Not } C_S$ 
  using confl by auto
have cdclW-all-struct-inv  $T$ 
  using cdclW.simps cdclW-all-struct-inv-inv confl inv by blast
then have cdclW-M-level-inv  $T$ 
  unfolding cdclW-all-struct-inv-def by auto
then consider
  (no-bt) skip-or-resolve**  $T$   $U$ 
  | (bt)  $T'$  where skip-or-resolve**  $T$   $T'$  and backtrack  $T' U$ 
  using bj rtrancp-cdclW-bj-skip-or-resolve-backtrack unfolding full-def by meson
then show ?case
proof cases
case no-bt
then have conflicting  $U \neq C\text{-True}$ 
  using confl by (induction rule: rtrancp-induct) auto
moreover then have no-step cdclW-merge  $U$ 
  by (auto simp: cdclW-merge.simps)
ultimately show ?thesis by blast
next
case bt note s-or-r = this(1) and bt = this(2)
have cdclW**  $T$   $T'$ 
  using s-or-r mono-rtrancp[of skip-or-resolve cdclW] rtrancp-skip-or-resolve-rtrancp-cdclW
  by blast
then have cdclW-M-level-inv  $T'$ 
  using rtrancp-cdclW-consistent-inv (cdclW-M-level-inv  $T'$ ) by blast
then obtain  $M1$   $M2$   $i$   $D$   $L$   $K$  where
  confl- $T'$ : conflicting  $T' = C\text{-Clause } (D + \{\#L\# \})$  and
   $M1$ - $M2$ : (Marked  $K$  ( $i+1$ )  $\#$   $M1$ ,  $M2$ )  $\in$  set (get-all-marked-decomposition (trail  $T'$ )) and
  get-level  $L$  (trail  $T'$ ) = backtrack-lvl  $T'$  and
  get-level  $L$  (trail  $T'$ ) = get-maximum-level ( $D + \{\#L\# \}$ ) (trail  $T'$ ) and
  get-maximum-level  $D$  (trail  $T'$ ) =  $i$  and
  undef- $L$ : undefined-lit  $M1$   $L$  and
   $U$ :  $U \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\# \}))$ 
    (reduce-trail-to  $M1$ 
      (add-learned-cls ( $D + \{\#L\# \}$ )
        (update-backtrack-lvl  $i$ 
          (update-conflicting  $C\text{-True } T'$ ))))
  using bt by (auto elim: backtrack-levE)
have [simp]: clauses  $S =$  clauses  $T$ 
  using confl by auto
have [simp]: clauses  $T =$  clauses  $T'$ 
  using s-or-r
proof (induction)
case base

```

```

    then show ?case by simp
next
  case (step U V) note st = this(1) and s-o-r = this(2) and IH = this(3)
  have clauses U = clauses V
    using s-o-r by auto
  then show ?case using IH by auto
qed
have inv-T: cdclW-all-struct-inv T
  by (meson cdclW-cp.simps confl inv r-into-rtrancpl rtrancpl-cdclW-all-struct-inv-inv
      rtrancpl-cdclW-cp-rtrancpl-cdclW)
have cdclW** T T'
  using rtrancpl-skip-or-resolve-rtrancpl-cdclW s-or-r by blast
have inv-T': cdclW-all-struct-inv T'
  using ⟨cdclW** T T'⟩ inv-T rtrancpl-cdclW-all-struct-inv-inv by blast
have inv-U: cdclW-all-struct-inv U
  using cdclW-merge-restart-cdclW confl fw-r-conflict inv local.bj
  rtrancpl-cdclW-all-struct-inv-inv by blast

have [simp]: init-clss S = init-clss T'
  using ⟨cdclW** T T'⟩ cdclW-init-clss confl cdclW-all-struct-inv-def conflict inv
  by (metis ⟨cdclW-M-level-inv T'⟩ rtrancpl-cdclW-init-clss)
then have atm-L: atm-of L ∈ atms-of-mu (clauses S)
  using inv-T' confl-T' unfolding cdclW-all-struct-inv-def no-strange-atm-def clauses-def
  by auto
obtain M where tr-T: trail T = M @ trail T'
  using s-or-r by (induction rule: rtrancpl-induct) auto
obtain M' where
  tr-T': trail T' = M' @ Marked K (i+1) # tl (trail U) and
  tr-U: trail U = Propagated L (D + {#L#}) # tl (trail U)
  using U M1-M2 undef-L inv-T' unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
  by auto
def M'' ≡ M @ M'
  have tr-T: trail S = M'' @ Marked K (i+1) # tl (trail U)
  using tr-T tr-T' confl unfolding M''-def by auto
have init-clss T' + learned-clss S ⊢pm D + {#L#}
  using inv-T' confl-T' unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def clauses-def
  by simp
have reduce-trail-to (convert-trail-from-NOT (convert-trail-from-W M1)) S =
  reduce-trail-to M1 S
  by (rule reduce-trail-to-length) simp
moreover have trail (reduce-trail-to M1 S) = M1
  apply (rule reduce-trail-to-skip-beginning[of - M @ - @ M2 @ [Marked K (Suc i)]])
  using confl M1-M2 ⟨trail T = M @ trail T'⟩
  apply (auto dest!: get-all-marked-decomposition-exists-prepend
      elim!: conflictE)
  by (rule sym) auto
ultimately have [simp]: trail (reduce-trail-toNOT (convert-trail-from-W M1) S) = M1
  using M1-M2 confl by (auto simp add: reduce-trail-toNOT-reduce-trail-convert)
have every-mark-is-a-conflict U
  using inv-U unfolding cdclW-all-struct-inv-def cdclW-conflicting-def by simp
then have tl (trail U) ⊢as CNot D
  by (metis add-diff-cancel-left' append-self-conv2 tr-U union-commute)
have backjump-l S U
  apply (rule backjump-l[of - - - - L])
  using tr-T apply simp

```

```

    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def apply simp
    using U M1-M2 confl undef-L M1-M2 inv-T' inv unfolding cdclW-all-struct-inv-def
    cdclW-M-level-inv-def apply (auto simp: state-eqNOT-def simp del: state-simpNOT)[]
    using CS apply simp
    using tr-S-CS apply simp

    using U undef-L M1-M2 inv-T' inv unfolding cdclW-all-struct-inv-def
    cdclW-M-level-inv-def apply auto[]
    using undef-L atm-L apply simp
    using ⟨init-clss T' + learned-clss S ⊨pm D + {#L#}⟩ unfolding clauses-def apply simp
    apply (metis ⟨tl (trail U) ⊨as CNot D⟩ convert-trail-from-W-tl
    convert-trail-from-W-true-annots)
    using inv-T' inv-U U confl-T' undef-L M1-M2 unfolding cdclW-all-struct-inv-def
    distinct-cdclW-state-def by (simp add: cdclW-M-level-inv-decomp)
    then show ?thesis using cdclNOT-merged-bj-learn-backjump-l by fast
  qed
qed

abbreviation cdclNOT-restart where
cdclNOT-restart ≡ restart-ops.cdclNOT-raw-restart cdclNOT restart

lemma cdclW-merge-restart-is-cdclNOT-merged-bj-learn-restart-no-step:
assumes
  inv: cdclW-all-struct-inv S and
  cdclW: cdclW-merge-restart S T
shows cdclNOT-restart** S T ∨ (no-step cdclW-merge T ∧ conflicting T ≠ C-True)
proof –
consider
  (fw) cdclW-merge S T
  | (fw-r) restart S T
using cdclW by (meson cdclW-merge-restart.simps cdclW-rf.cases fw-conflict fw-decide fw-forget
  fw-propagate)
then show ?thesis
proof cases
  case fw
then have IH: cdclNOT-merged-bj-learn S T ∨ (no-step cdclW-merge T ∧ conflicting T ≠ C-True)
  using inv cdclW-merge-is-cdclNOT-merged-bj-learn by blast
have invS: invNOT S
  using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
have ff2: cdclNOT++ S T ⟶ cdclNOT** S T
  by (meson tranclp-into-rtranclp)
have ff3: no-dup ((convert-trail-from-W ∘ trail) S)
  using invS by simp
have cdclNOT ≤ cdclNOT-restart
  by (auto simp: restart-ops.cdclNOT-raw-restart.simps)
then show ?thesis
  using ff3 ff2 IH cdclNOT-merged-bj-learn-is-tranclp-cdclNOT
  rtranclp-mono[of cdclNOT cdclNOT-restart] invS predicate2D by blast
next
  case fw-r
then show ?thesis by (blast intro: restart-ops.cdclNOT-raw-restart.intros)
  qed
qed

```

**abbreviation**  $\mu_{FW} :: 'st \Rightarrow nat$  **where**

$\mu_{FW} S \equiv (\text{if no-step } cdcl_W\text{-merge } S \text{ then } 0 \text{ else } 1 + \mu_{CDCL}'\text{-merged } (\text{set-mset } (\text{init-clss } S)) S)$

**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**  $\mu_{FW} T < \mu_{FW} S$

**proof** –

**let** *?A* = *init-clss* *S*

**have** *atm-clauses*: *atms-of-mu* (*clauses* *S*)  $\subseteq$  *atms-of-mu* *?A*

**using** *inv* **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* *no-strange-atm-def* *clauses-def* **by** *auto*

**have** *atm-trail*: *atm-of* ‘*lits-of* (*trail* *S*)  $\subseteq$  *atms-of-mu* *?A*

**using** *inv* **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* *no-strange-atm-def* *clauses-def* **by** *auto*

**have** *n-d*: *no-dup* (*trail* *S*)

**using** *inv* **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** (*auto simp*: *cdcl<sub>W</sub>-M-level-inv-decomp*)

**have** [*simp*]:  $\neg$  *no-step* *cdcl<sub>W</sub>-merge* *S*

**using** *fw* **by** *auto*

**have** [*simp*]: *init-clss* *S* = *init-clss* *T*

**using** *cdcl<sub>W</sub>-merge-restart-cdcl<sub>W</sub>*[*of* *S* *T*] *inv* *rtranclp-cdcl<sub>W</sub>-init-clss*

**unfolding** *cdcl<sub>W</sub>-all-struct-inv-def*

**by** (*meson* *cdcl<sub>W</sub>-merge.simps* *cdcl<sub>W</sub>-merge-restart.simps* *cdcl<sub>W</sub>-rf.simps* *fw*)

**consider**

(*merged*) *cdcl<sub>NOT</sub>-merged-bj-learn* *S* *T*

| (*n-s*) *no-step* *cdcl<sub>W</sub>-merge* *T*

**using** *cdcl<sub>W</sub>-merge-is-cdcl<sub>NOT</sub>-merged-bj-learn* *inv* *fw* **by** *blast*

**then show** *?thesis*

**proof** *cases*

**case** *merged*

**then show** *?thesis*

**using** *cdcl<sub>NOT</sub>-decreasing-measure*‘[*OF* - - *atm-clauses*] *atm-trail* *n-d*

**by** (*auto split*: *split-if*)

**next**

**case** *n-s*

**then show** *?thesis* **by** *simp*

**qed**

**qed**

**lemma** *wf-cdcl<sub>W</sub>-merge*: *wf*  $\{(T, S). \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T\}$

**apply** (*rule* *wfP-if-measure*[*of* - -  $\mu_{FW}$ ])

**using** *cdcl<sub>W</sub>-merge- $\mu_{FW}$ -decreasing* **by** *blast*

**lemma** *cdcl<sub>W</sub>-all-struct-inv-tranclp-cdcl<sub>W</sub>-merge-tranclp-cdcl<sub>W</sub>-merge-cdcl<sub>W</sub>-all-struct-inv*:

**assumes**

*inv*: *cdcl<sub>W</sub>-all-struct-inv* *b*

*cdcl<sub>W</sub>-merge*<sup>++</sup> *b* *a*

**shows**  $(\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T)^{++} b a$

**using** *assms*(2)

**proof** *induction*

**case** *base*

**then show** *?case* **using** *inv* **by** *auto*

**next**

**case** (*step c d*) **note** *st* = *this*(1) **and** *fw* = *this*(2) **and** *IH* = *this*(3)

**have** *cdcl<sub>W</sub>-all-struct-inv* *c*

**using** *tranclp-into-rtranclp*[*OF* *st*] *cdcl<sub>W</sub>-merge-rtranclp-cdcl<sub>W</sub>*

*assms*(1) *rtranclp-cdcl<sub>W</sub>-all-struct-inv-inv* *rtranclp-mono*[*of* *cdcl<sub>W</sub>-merge* *cdcl<sub>W</sub>*<sup>\*\*</sup>] **by** *fastforce*

then have  $(\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T)^{++} c d$   
 using *fw* by *auto*  
 then show *?case* using *IH* by *auto*  
 qed

**lemma** *wf-trancl-cdcl<sub>W</sub>-merge*: *wf*  $\{(T, S). \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge}^{++} S T\}$   
 using *wf-trancl[OF wf-cdcl<sub>W</sub>-merge]*  
 apply (*rule wf-subset*)  
 by (*auto simp: trancl-set-trancl*  
*cdcl<sub>W</sub>-all-struct-inv-trancl-cdcl<sub>W</sub>-merge-trancl-cdcl<sub>W</sub>-merge-cdcl<sub>W</sub>-all-struct-inv*)

**lemma** *backtrack-is-full1-cdcl<sub>W</sub>-bj*:  
 assumes *bt*: *backtrack* *S T* and *inv*: *cdcl<sub>W</sub>-M-level-inv* *S*  
 shows *full1 cdcl<sub>W</sub>-bj* *S T*

**proof** –  
 have *no-step cdcl<sub>W</sub>-bj* *T*  
 using *bt inv backtrack-no-cdcl<sub>W</sub>-bj* by *blast*  
 moreover have *cdcl<sub>W</sub>-bj*<sup>++</sup> *S T*  
 using *bt* by *auto*  
 ultimately show *?thesis* unfolding *full1-def* by *blast*  
 qed

**lemma** *rtrancl-cdcl<sub>W</sub>-conflicting-true-cdcl<sub>W</sub>-merge-restart*:  
 assumes *cdcl<sub>W</sub>*<sup>\*\*</sup> *S V* and *inv*: *cdcl<sub>W</sub>-M-level-inv* *S* and *conflicting* *S = C-True*  
 shows  $(\text{cdcl}_W\text{-merge-restart}^{**} S V \wedge \text{conflicting } V = C\text{-True})$   
 $\vee (\exists T U. \text{cdcl}_W\text{-merge-restart}^{**} S T \wedge \text{conflicting } V \neq C\text{-True} \wedge \text{conflict } T U \wedge \text{cdcl}_W\text{-bj}^{**} U V)$   
 using *assms*

**proof** *induction*

case *base*

then show *?case* by *simp*

next

case (*step U V*) note *st = this(1)* and *cdcl<sub>W</sub> = this(2)* and *IH = this(3)[OF this(4–)]* and  
*cnfl[*simp*] = this(5)* and *inv = this(4)*

from *cdcl<sub>W</sub>*

show *?case*

**proof** (*cases*)

case *propagate*

moreover then have *conflicting U = C-True*

by *auto*

moreover have *conflicting V = C-True*

using *propagate* by *auto*

ultimately show *?thesis* using *IH cdcl<sub>W</sub>-merge-restart.fw-r-propagate[of U V]* by *auto*

next

case *conflict*

moreover then have *conflicting U = C-True*

by *auto*

moreover have *conflicting V ≠ C-True*

using *conflict* by *auto*

ultimately show *?thesis* using *IH* by *auto*

next

case *other*

then show *?thesis*

**proof** *cases*

case *decide*

moreover then have *conflicting U = C-True*

```

    by auto
  ultimately show ?thesis using IH cdclW-merge-restart.fw-r-decide[of U V] by auto
next
case bj
moreover {
  assume skip-or-resolve U V
  have f1: cdclW-bj++ U V
  by (simp add: local.bj tranclp.r-into-trancl)
  obtain T T' :: 'st where
    f2: cdclW-merge-restart** S U
    ∨ cdclW-merge-restart** S T ∧ conflicting U ≠ C-True
    ∧ conflict T T' ∧ cdclW-bj** T' U
  using IH confl by blast
  then have ?thesis
  proof -
    have conflicting V ≠ C-True ∧ conflicting U ≠ C-True
    using ⟨skip-or-resolve U V⟩ by auto
    then show ?thesis
    by (metis (no-types) IH f1 rtranclp-trans tranclp-into-rtranclp)
  qed
}
moreover {
  assume backtrack U V
  then have conflicting U ≠ C-True by auto
  then obtain T T' where
    cdclW-merge-restart** S T and
    conflicting U ≠ C-True and
    conflict T T' and
    cdclW-bj** T' U
  using IH confl by blast
  have invU: cdclW-M-level-inv U
  using inv rtranclp-cdclW-consistent-inv step.hyps(1) by blast
  then have conflicting V = C-True
  using ⟨backtrack U V⟩ inv by (auto elim: backtrack-levE
    simp: cdclW-M-level-inv-decomp)
  have full cdclW-bj T' V
  apply (rule rtranclp-fullI[of cdclW-bj T' U V])
  using ⟨cdclW-bj** T' U⟩ apply fast
  using ⟨backtrack U V⟩ backtrack-is-full1-cdclW-bj invU unfolding full1-def full-def
  by blast
  then have ?thesis
  using cdclW-merge-restart.fw-r-conflict[of T T' V] ⟨conflict T T'⟩
  ⟨cdclW-merge-restart** S T⟩ ⟨conflicting V = C-True⟩ by auto
}
ultimately show ?thesis by (auto simp: cdclW-bj.simps)
qed
next
case rf
moreover then have conflicting U = C-True and conflicting V = C-True
  by (auto simp: cdclW-rf.simps)
ultimately show ?thesis using IH cdclW-merge-restart.fw-r-rf[of U V] by auto
qed
qed

```

**lemma** no-step-cdcl<sub>W</sub>-no-step-cdcl<sub>W</sub>-merge-restart: no-step cdcl<sub>W</sub> S  $\implies$  no-step cdcl<sub>W</sub>-merge-restart

$S$

by (auto simp:  $cdcl_W.simps$   $cdcl_W$ -merge-restart.simps  $cdcl_W$ -o.simps  $cdcl_W$ -bj.simps)

**lemma**  $no\text{-}step\text{-}cdcl_W\text{-merge-restart-no-step-cdcl_W$ :

**assumes**

$conflicting\ S = C\text{-}True$  **and**

$cdcl_W\text{-}M\text{-level-inv}\ S$  **and**

$no\text{-}step\ cdcl_W\text{-merge-restart}\ S$

**shows**  $no\text{-}step\ cdcl_W\ S$

**proof** –

{ **fix**  $S'$

**assume**  $conflict\ S\ S'$

**then have**  $cdcl_W\ S\ S'$  **using**  $cdcl_W.conflict$  **by** *auto*

**then have**  $cdcl_W\text{-}M\text{-level-inv}\ S'$

**using**  $assms(2)\ cdcl_W\text{-consistent-inv}$  **by** *blast*

**then obtain**  $S''$  **where**  $full\ cdcl_W\text{-bj}\ S'\ S''$

**using**  $cdcl_W\text{-bj-exists-normal-form}[of\ S']$  **by** *auto*

**then have** *False*

**using**  $\langle conflict\ S\ S' \rangle\ assms(3)\ fw\text{-}r\text{-}conflict$  **by** *blast*

}

**then show** *?thesis*

**using**  $assms$  **unfolding**  $cdcl_W.simps$   $cdcl_W$ -merge-restart.simps  $cdcl_W$ -o.simps  $cdcl_W$ -bj.simps

**by** *fastforce*

**qed**

**lemma**  $rtranclp\text{-}cdcl_W\text{-merge-restart-no-step-cdcl_W}\text{-bj}$ :

**assumes**

$cdcl_W\text{-merge-restart}^{**}\ S\ T$  **and**

$conflicting\ S = C\text{-}True$

**shows**  $no\text{-}step\ cdcl_W\text{-bj}\ T$

**using**  $assms$

**by** (*induction rule*:  $rtranclp\text{-}induct$ )

(*fastforce*  $simp$ :  $cdcl_W\text{-bj.simps}$   $cdcl_W$ -rf.simps  $cdcl_W$ -merge-restart.simps  $full\text{-}def$ ) +

If  $conflicting\ S \neq C\text{-}True$ , 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**  $conflicting\text{-}true\text{-}full\text{-}cdcl_W\text{-iff-full-cdcl_W}\text{-merge}$ :

**assumes**  $confl$ :  $conflicting\ S = C\text{-}True$  **and**  $lev$ :  $cdcl_W\text{-}M\text{-level-inv}\ S$

**shows**  $full\ cdcl_W\ S\ V \iff full\ cdcl_W\text{-merge-restart}\ S\ V$

**proof**

**assume**  $full$ :  $full\ cdcl_W\text{-merge-restart}\ S\ V$

**then have**  $st$ :  $cdcl_W^{**}\ S\ V$

**using**  $rtranclp\text{-}mono[of\ cdcl_W\text{-merge-restart}\ cdcl_W^{**}]$   $cdcl_W\text{-merge-restart-cdcl_W}$

**unfolding**  $full\text{-}def$  **by** *auto*

**have**  $n\text{-}s$ :  $no\text{-}step\ cdcl_W\text{-merge-restart}\ V$

**using**  $full$  **unfolding**  $full\text{-}def$  **by** *auto*

**have**  $n\text{-}s\text{-}bj$ :  $no\text{-}step\ cdcl_W\text{-bj}\ V$

**using**  $rtranclp\text{-}cdcl_W\text{-merge-restart-no-step-cdcl_W}\text{-bj}\ confl\ full$  **unfolding**  $full\text{-}def$  **by** *auto*

**have**  $\bigwedge S'.\ conflict\ V\ S' \implies cdcl_W\text{-}M\text{-level-inv}\ S'$

**using**  $cdcl_W.conflict\ cdcl_W\text{-consistent-inv}\ lev\ rtranclp\text{-}cdcl_W\text{-consistent-inv}\ st$  **by** *blast*

**then have**  $\bigwedge S'.\ conflict\ V\ S' \implies False$

**using**  $n\text{-}s\ n\text{-}s\text{-}bj\ cdcl_W\text{-bj-exists-normal-form}\ cdcl_W\text{-merge-restart.simps}$  **by** *meson*

**then have**  $n\text{-}s\text{-}cdcl_W$ :  $no\text{-}step\ cdcl_W\ V$

```

    using n-s n-s-bj by (auto simp: cdclW.simps cdclW-o.simps cdclW-merge-restart.simps)
  then show full cdclW S V using st unfolding full-def by auto
next
assume full: full cdclW S V
have no-step cdclW-merge-restart V
  using full no-step-cdclW-no-step-cdclW-merge-restart unfolding full-def by blast
moreover
consider
  (fw) cdclW-merge-restart** S V and conflicting V = C-True
| (bj) T U where
  cdclW-merge-restart** S T and
  conflicting V ≠ C-True and
  conflict T U and
  cdclW-bj** U V
using full rtrancl-cdclW-conflicting-true-cdclW-merge-restart confl lev unfolding full-def
by meson
then have cdclW-merge-restart** S V
proof cases
  case fw
  then show ?thesis by fast
next
  case (bj T U)
  have no-step cdclW-bj V
    by (meson cdclW-o.bj full full-def other)
  then have full cdclW-bj U V
    using ⟨ cdclW-bj** U V ⟩ unfolding full-def by auto
  then have cdclW-merge-restart T V
    using ⟨ conflict T U ⟩ cdclW-merge-restart.fw-r-conflict by blast
  then show ?thesis using ⟨ cdclW-merge-restart** S T ⟩ by auto
qed
ultimately show full cdclW-merge-restart S V unfolding full-def by fast
qed

```

**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 ⟷ full cdcl<sub>W</sub>-merge-restart (init-state N) V*  
 by (rule *conflicting-true-full-cdcl<sub>W</sub>-iff-full-cdcl<sub>W</sub>-merge*) auto

## 19.4 FW with strategy

### 19.4.1 The intermediate step

**inductive** *cdcl<sub>W</sub>-s' :: 'st ⇒ 'st ⇒ bool* where  
*conflict': full1 cdcl<sub>W</sub>-cp S S' ⟹ cdcl<sub>W</sub>-s' S S' |*  
*decide': decide S S' ⟹ no-step cdcl<sub>W</sub>-cp S ⟹ full cdcl<sub>W</sub>-cp S' S'' ⟹ cdcl<sub>W</sub>-s' S S'' |*  
*bj': full1 cdcl<sub>W</sub>-bj S S' ⟹ no-step cdcl<sub>W</sub>-cp S ⟹ full cdcl<sub>W</sub>-cp S' S'' ⟹ cdcl<sub>W</sub>-s' S S''*

**inductive-cases** *cdcl<sub>W</sub>-s'E: cdcl<sub>W</sub>-s' S T*

**lemma** *rtranclp-cdcl<sub>W</sub>-bj-full1-cdclp-cdcl<sub>W</sub>-stgy:*  
*cdcl<sub>W</sub>-bj\*\* S S' ⟹ full cdcl<sub>W</sub>-cp S' S'' ⟹ cdcl<sub>W</sub>-stgy\*\* S S''*  
**proof** (induction rule: *converse-rtranclp-induct*)  
 case base  
 then show *?case* by (metis *cdcl<sub>W</sub>-stgy.conflict' full-unfold rtranclp.simps*)  
next  
 case (step *T U*) note *st = this(2)* and *bj = this(1)* and *IH = this(3)[OF this(4)]*  
 have *no-step cdcl<sub>W</sub>-cp T*



```

    using bj by (auto simp add: cdclW-bj.simps)
consider
  (U) U = S'
| (U') U' where cdclW-bj U U' and cdclW-bj** U' S'
  using st by (metis converse-rtranclpE)
then show ?case
proof cases
  case U
  then show ?thesis
    using ⟨no-step cdclW-cp T⟩ cdclW-o.bj local.bj other' step.prem by (meson r-into-rtranclp)
next
  case U' note U' = this(1)
  have no-step cdclW-cp U
    using U' by (fastforce simp: cdclW-cp.simps cdclW-bj.simps)
  then have full cdclW-cp U U
    by (simp add: full-unfold)
  then have cdclW-stgy T U
    using ⟨no-step cdclW-cp T⟩ cdclW-stgy.simps local.bj cdclW-o.bj by meson
  then show ?thesis using IH by auto
qed
qed

lemma cdclW-s'-is-rtranclp-cdclW-stgy:
  cdclW-s' S T  $\implies$  cdclW-stgy** S T
  apply (induction rule: cdclW-s'.induct)
  apply (auto intro: cdclW-stgy.intros)[]
  apply (meson decide other' r-into-rtranclp)
  by (metis full1-def rtranclp-cdclW-bj-full1-cdclp-cdclW-stgy tranclp-into-rtranclp)

lemma cdclW-cp-cdclW-bj-bissimulation:
  assumes
    full cdclW-cp T U and
    cdclW-bj** T T' and
    cdclW-all-struct-inv T and
    no-step cdclW-bj T'
  shows full cdclW-cp T' U
     $\vee (\exists U'' U''. \text{full cdcl}_W\text{-cp } T' U'' \wedge \text{full1 cdcl}_W\text{-bj } U U' \wedge \text{full cdcl}_W\text{-cp } U' U'' \wedge \text{cdcl}_W\text{-s'}^{**} U U'')$ 
  using assms(2,1,3,4)
proof (induction rule: rtranclp-induct)
  case base
  then show ?case by blast
next
  case (step T' T'') note st = this(1) and bj = this(2) and IH = this(3)[OF this(4,5)] and
    full = this(4) and inv = this(5)
  have cdclW** T T''
    by (metis (no-types, lifting) cdclW-o.bj local.bj mono-rtranclp[of cdclW-bj cdclW T T''] other
    st rtranclp.rtrancl-into-rtrancl)
  then have inv-T'': cdclW-all-struct-inv T''
    using inv rtranclp-cdclW-all-struct-inv-inv by blast
  have cdclW-bj++ T T''
    using local.bj st by auto
  have full1 cdclW-bj T T''
    by (metis ⟨cdclW-bj++ T T'⟩ full1-def step.prem(3))
  then have T = U
  proof -

```

```

obtain  $Z$  where  $cdcl_W\text{-}bj\ T\ Z$ 
  by ( $meson\ tranclpD\ \langle cdcl_W\text{-}bj^{++}\ T\ T'' \rangle$ )
{ assume  $cdcl_W\text{-}cp^{++}\ T\ U$ 
  then obtain  $Z'$  where  $cdcl_W\text{-}cp\ T\ Z'$ 
    by ( $meson\ tranclpD$ )
    then have  $False$ 
    using  $\langle cdcl_W\text{-}bj\ T\ Z \rangle$  by ( $fastforce\ simp:\ cdcl_W\text{-}bj.simps\ cdcl_W\text{-}cp.simps$ )
  }
then show  $?thesis$ 
  using  $full\ unfolding\ full\text{-}def\ rtranclp\text{-}unfold$  by  $blast$ 
qed
obtain  $U''$  where  $full\ cdcl_W\text{-}cp\ T''\ U''$ 
  using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv\ inv\text{-}T''$  by  $blast$ 
moreover then have  $cdcl_W\text{-}stgy^{**}\ U\ U''$ 
  by ( $metis\ \langle T = U \rangle\ \langle cdcl_W\text{-}bj^{++}\ T\ T'' \rangle\ rtranclp\text{-}cdcl_W\text{-}bj\text{-}full1\text{-}cdclp\text{-}cdcl_W\text{-}stgy\ rtranclp\text{-}unfold$ )
moreover have  $cdcl_W\text{-}s^{**}\ U\ U''$ 
proof –
  obtain  $ss :: 'st \Rightarrow 'st$  where
     $f1: \forall x2. (\exists v3. cdcl_W\text{-}cp\ x2\ v3) = cdcl_W\text{-}cp\ x2\ (ss\ x2)$ 
    by  $moura$ 
  have  $\neg cdcl_W\text{-}cp\ U\ (ss\ U)$ 
    by ( $meson\ full\ full\text{-}def$ )
  then show  $?thesis$ 
    using  $f1$  by ( $metis\ (no\text{-}types)\ \langle T = U \rangle\ \langle full1\ cdcl_W\text{-}bj\ T\ T'' \rangle\ bj'\ calculation(1)\ r\text{-}into\text{-}rtranclp$ )
  qed
ultimately show  $?case$ 
  using  $\langle full1\ cdcl_W\text{-}bj\ T\ T'' \rangle\ \langle full\ cdcl_W\text{-}cp\ T''\ U'' \rangle$  unfolding  $\langle T = U \rangle$  by  $blast$ 
qed

```

**lemma**  $cdcl_W\text{-}cp\text{-}cdcl_W\text{-}bj\text{-}bissimulation'$ :

```

assumes
   $full\ cdcl_W\text{-}cp\ T\ U$  and
   $cdcl_W\text{-}bj^{**}\ T\ T'$  and
   $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$  and
   $no\text{-}step\ cdcl_W\text{-}bj\ T'$ 
shows  $full\ cdcl_W\text{-}cp\ T'\ U$ 
   $\vee (\exists U'. full1\ cdcl_W\text{-}bj\ U\ U' \wedge (\forall U''. full\ cdcl_W\text{-}cp\ U'\ U'' \longrightarrow full\ cdcl_W\text{-}cp\ T'\ U''$ 
     $\wedge cdcl_W\text{-}s^{**}\ U\ U''))$ 
using  $assms(2,1,3,4)$ 
proof ( $induction\ rule:\ rtranclp\text{-}induct$ )
  case  $base$ 
    then show  $?case$  by  $blast$ 
next
  case ( $step\ T'\ T''$ ) note  $st = this(1)$  and  $bj = this(2)$  and  $IH = this(3)[OF\ this(4,5)]$  and
     $full = this(4)$  and  $inv = this(5)$ 
  have  $cdcl_W^{**}\ T\ T''$ 
    by ( $metis\ (no\text{-}types,\ lifting)\ cdcl_W\text{-}o.bj\ local.bj\ mono\text{-}rtranclp[of\ cdcl_W\text{-}bj\ cdcl_W\ T\ T'']\ other\ st\ rtranclp.rtrancl\text{-}into\text{-}rtrancl$ )
  then have  $inv\text{-}T'': cdcl_W\text{-}all\text{-}struct\text{-}inv\ T''$ 
    using  $inv\ rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$  by  $blast$ 
  have  $cdcl_W\text{-}bj^{++}\ T\ T''$ 
    using  $local.bj\ st$  by  $auto$ 
  have  $full1\ cdcl_W\text{-}bj\ T\ T''$ 
    by ( $metis\ \langle cdcl_W\text{-}bj^{++}\ T\ T'' \rangle\ full1\text{-}def\ step.prem(3)$ )

```

```

then have  $T = U$ 
proof -
  obtain  $Z$  where  $cdcl_W\text{-bj } T Z$ 
  by (meson  $trancplD \langle cdcl_W\text{-bj}^{++} T T'' \rangle$ )
  { assume  $cdcl_W\text{-cp}^{++} T U$ 
    then obtain  $Z'$  where  $cdcl_W\text{-cp } T Z'$ 
    by (meson  $trancplD$ )
    then have False
    using  $\langle cdcl_W\text{-bj } T Z \rangle$  by (fastforce  $simp: cdcl_W\text{-bj.simps } cdcl_W\text{-cp.simps}$ )
  }
  then show ?thesis
  using full unfolding full-def rtrancpl-unfold by blast
qed
{ fix  $U''$ 
  assume full  $cdcl_W\text{-cp } T'' U''$ 
  moreover then have  $cdcl_W\text{-stgy}^{**} U U''$ 
  by (metis  $\langle T = U \rangle \langle cdcl_W\text{-bj}^{++} T T'' \rangle$  rtrancpl-cdcl_W-bj-full1-cdclp-cdcl_W-stgy rtrancpl-unfold)
  moreover have  $cdcl_W\text{-s}^{**} U U''$ 
  proof -
    obtain  $ss :: 'st \Rightarrow 'st$  where
       $f1: \forall x2. (\exists v3. cdcl_W\text{-cp } x2 v3) = cdcl_W\text{-cp } x2 (ss x2)$ 
    by moura
    have  $\neg cdcl_W\text{-cp } U (ss U)$ 
    by (meson  $assms(1)$  full-def)
    then show ?thesis
    using  $f1$  by (metis (no-types)  $\langle T = U \rangle \langle full1 cdcl_W\text{-bj } T T'' \rangle$  bj' calculation(1)
      r-into-rtrancpl)
  qed
  ultimately have full1  $cdcl_W\text{-bj } U T''$  and  $cdcl_W\text{-s}^{**} T'' U''$ 
  using  $\langle full1 cdcl_W\text{-bj } T T'' \rangle \langle full cdcl_W\text{-cp } T'' U'' \rangle$  unfolding  $\langle T = U \rangle$ 
  apply blast
  by (metis  $\langle full cdcl_W\text{-cp } T'' U'' \rangle$   $cdcl_W\text{-s'.simps full-unfold rtrancpl.simps}$ )
}
then show ?case
using  $\langle full1 cdcl_W\text{-bj } T T'' \rangle$  full bj' unfolding  $\langle T = U \rangle$  full-def by (metis r-into-rtrancpl)
qed

lemma cdcl_W-stgy-cdcl_W-s'-connected:
  assumes  $cdcl_W\text{-stgy } S U$  and  $cdcl_W\text{-all-struct-inv } S$ 
  shows  $cdcl_W\text{-s}' S U$ 
   $\vee (\exists U'. full1 cdcl_W\text{-bj } U U' \wedge (\forall U''. full cdcl_W\text{-cp } U' U'' \longrightarrow cdcl_W\text{-s}' S U''))$ 
  using assms
proof (induction rule: cdcl_W-stgy.induct)
  case (conflict'  $T$ )
  then have  $cdcl_W\text{-s}' S T$ 
  using  $cdcl_W\text{-s'}.conflict'$  by blast
  then show ?case
  by blast
next
  case (other'  $T U$ ) note  $o = this(1)$  and  $n\text{-s} = this(2)$  and  $full = this(3)$  and  $inv = this(4)$ 
  show ?case
  using  $o$ 
  proof cases
    case decide
    then show ?thesis using  $cdcl_W\text{-s'.simps full n-s}$  by blast
  end
end

```

```

next
  case bj
  have inv-T: cdclW-all-struct-inv T
    using cdclW-all-struct-inv-inv o other other'.prems by blast
  consider
    (cp) full cdclW-cp T U and no-step cdclW-bj T
  | (fbj) T' where full1 cdclW-bj T T'
  apply (cases no-step cdclW-bj T)
    using full apply blast
  using cdclW-bj-exists-normal-form[of T] inv-T unfolding cdclW-all-struct-inv-def
  by (metis full-unfold)
then show ?thesis
  proof cases
    case cp
    then show ?thesis
      proof -
        obtain ss :: 'st ⇒ 'st where
          f1: ∀ s sa sb. (¬ full1 cdclW-bj s sa ∨ cdclW-cp s (ss s) ∨ ¬ full cdclW-cp sa sb)
            ∨ cdclW-s' s sb
        using bj' by moura
        have full1 cdclW-bj S T
          by (simp add: cp(2) full1-def local.bj tranclp.r-into-trancl)
        then show ?thesis
          using f1 full n-s by blast
      qed
    next
      case (fbj U')
      then have full1 cdclW-bj S U'
        using bj unfolding full1-def by auto
      moreover have no-step cdclW-cp S
        using n-s by blast
      moreover have T = U
        using full fbj unfolding full1-def full-def rtranclp-unfold
        by (force dest!: tranclpD simp:cdclW-bj.simps)
      ultimately show ?thesis using cdclW-s'.bj'[of S U'] using fbj by blast
    qed
  qed
qed

lemma cdclW-stgy-cdclW-s'-connected':
  assumes cdclW-stgy S U and cdclW-all-struct-inv S
  shows cdclW-s' S U
    ∨ (∃ U' U''. cdclW-s' S U'' ∧ full1 cdclW-bj U U' ∧ full cdclW-cp U' U'')
  using assms
proof (induction rule: cdclW-stgy.induct)
  case (conflict' T)
  then have cdclW-s' S T
    using cdclW-s'.conflict' by blast
  then show ?case
    by blast
next
  case (other' T U) note o = this(1) and n-s = this(2) and full = this(3) and inv = this(4)
  show ?case
    using o
  proof cases

```

```

case decide
then show ?thesis using cdclW-s'.simps full n-s by blast
next
case bj
have cdclW-all-struct-inv T
  using cdclW-all-struct-inv-inv o other other'.prems by blast
then obtain T' where T': full cdclW-bj T T'
  using cdclW-bj-exists-normal-form unfolding full-def cdclW-all-struct-inv-def by metis
then have full cdclW-bj S T'
  proof –
    have f1: cdclW-bj** T T' ∧ no-step cdclW-bj T'
      by (metis (no-types) T' full-def)
    then have cdclW-bj** S T'
      by (meson converse-rtranclp-into-rtranclp local.bj)
    then show ?thesis
      using f1 by (simp add: full-def)
  qed
have cdclW-bj** T T'
  using T' unfolding full-def by simp
have cdclW-all-struct-inv T
  using cdclW-all-struct-inv-inv o other other'.prems by blast
then consider
  (T'U) full cdclW-cp T' U
  | (U) U' U'' where
    full cdclW-cp T' U'' and
    full1 cdclW-bj U U' and
    full cdclW-cp U' U'' and
    cdclW-s*** U U''
  using cdclW-cp-cdclW-bj-bissimulation[OF full ⟨cdclW-bj** T T'⟩ T' unfolding full-def
  by blast
then show ?thesis by (metis T' cdclW-s'.simps full-full1 local.bj n-s)
qed
qed

```

**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*  
**using** *cdcl<sub>W</sub>-stgy-cdcl<sub>W</sub>-s'-connected[OF assms(1,2)] assms(3)*  
**by** (*metis (no-types, lifting) full1-def tranclpD*)

**lemma** *rtranclp-cdcl<sub>W</sub>-stgy-connected-to-rtranclp-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 ∨ (∃ T. cdcl<sub>W</sub>-s\*\*\* S T ∧ cdcl<sub>W</sub>-bj<sup>++</sup> T U ∧ conflicting U ≠ C-True)*  
**using** *assms(1)*  
**proof** *induction*  
**case** *base*  
**then show** ?case **by** *simp*  
**next**  
**case** (*step T V*) **note** *st = this(1) and o = this(2) and IH = this(3)*  
**from** *o* **show** ?case  
**proof** *cases*  
**case** *conflict'*  
**then have** *f2: cdcl<sub>W</sub>-s' T V*  
**using** *cdcl<sub>W</sub>-s'.conflict'* **by** *blast*  
**obtain** *ss :: 'st* **where**

```

    f3:  $S = T \vee \text{cdcl}_W\text{-stgy}^{**} S \text{ ss} \wedge \text{cdcl}_W\text{-stgy ss } T$ 
    by (metis (full-types) rtranclp.simps st)
  obtain ssa :: 'st where
    cdclW-cp T ssa
    using conflict' by (metis (no-types) full1-def tranclpD)
  then have  $S = T$ 
    using f3 by (metis (no-types) cdclW-stgy.simps full-def full1-def)
  then show ?thesis
    using f2 by blast
next
case (other' U) note o = this(1) and n-s = this(2) and full = this(3)
then show ?thesis
  using o
  proof (cases rule: cdclW-o-rule-cases)
    case decide
    then have cdclW-s'** S T
      using IH by auto
    then show ?thesis
      by (meson decide decide' full n-s rtranclp.rtrancl-into-rtrancl)
  next
  case backtrack
  consider
    (s') cdclW-s'** S T
  | (bj) S' where cdclW-s'** S S' and cdclW-bj++ S' T and conflicting T ≠ C-True
    using IH by blast
  then show ?thesis
    proof cases
      case s'
      moreover
        have cdclW-M-level-inv T
          using inv local.step(1) rtranclp-cdclW-stgy-consistent-inv by auto
        then have full1 cdclW-bj T U
          using backtrack-is-full1-cdclW-bj backtrack by blast
        then have cdclW-s' T V
          using full bj' n-s by blast
        ultimately show ?thesis by auto
    next
    case (bj S') note S-S' = this(1) and bj-T = this(2)
    have no-step cdclW-cp S'
      using bj-T by (fastforce simp: cdclW-cp.simps cdclW-bj.simps dest!: tranclpD)
    moreover
      have cdclW-M-level-inv T
        using inv local.step(1) rtranclp-cdclW-stgy-consistent-inv by auto
      then have full1 cdclW-bj T U
        using backtrack-is-full1-cdclW-bj backtrack by blast
      then have full1 cdclW-bj S' U
        using bj-T unfolding full1-def by fastforce
      ultimately have cdclW-s' S' V using full by (simp add: bj')
      then show ?thesis using S-S' by auto
    qed
  next
  case skip
  then have [simp]: U = V
    using full converse-rtranclpE unfolding full-def by fastforce

```

```

consider
  (s')  $cdcl_W-s'^{**} S T$ 
  | (bj)  $S'$  where  $cdcl_W-s'^{**} S S'$  and  $cdcl_W-bj^{++} S' T$  and conflicting  $T \neq C\text{-True}$ 
  using IH by blast
then show ?thesis
proof cases
  case s'
  have  $cdcl_W-bj^{++} T V$ 
  using skip by force
  moreover have conflicting  $V \neq C\text{-True}$ 
  using skip by auto
  ultimately show ?thesis using s' by auto
next
  case (bj S') note  $S-S' = this(1)$  and  $bj-T = this(2)$ 
  have  $cdcl_W-bj^{++} S' V$ 
  using skip  $bj-T$  by (metis  $\langle U = V \rangle cdcl_W-bj.skip tranclp.simps$ )

  moreover have conflicting  $V \neq C\text{-True}$ 
  using skip by auto
  ultimately show ?thesis using  $S-S'$  by auto
qed
next
  case resolve
  then have [simp]:  $U = V$ 
  using full converse-rtranclpE unfolding full-def by fastforce
  consider
    (s')  $cdcl_W-s'^{**} S T$ 
    | (bj)  $S'$  where  $cdcl_W-s'^{**} S S'$  and  $cdcl_W-bj^{++} S' T$  and conflicting  $T \neq C\text{-True}$ 
    using IH by blast
  then show ?thesis
  proof cases
    case s'
    have  $cdcl_W-bj^{++} T V$ 
    using resolve by force
    moreover have conflicting  $V \neq C\text{-True}$ 
    using resolve by auto
    ultimately show ?thesis using s' by auto
  next
    case (bj S') note  $S-S' = this(1)$  and  $bj-T = this(2)$ 
    have  $cdcl_W-bj^{++} S' V$ 
    using resolve  $bj-T$  by (metis  $\langle U = V \rangle cdcl_W-bj.resolve tranclp.simps$ )
    moreover have conflicting  $V \neq C\text{-True}$ 
    using resolve by auto
    ultimately show ?thesis using  $S-S'$  by auto
  qed
qed
qed
qed

lemma n-step-cdcl_W-stgy-iff-no-step-cdcl_W-cl-cdcl_W-o:
  assumes inv:  $cdcl_W\text{-all-struct-inv } S$ 
  shows  $no\text{-step } cdcl_W-s' S \longleftrightarrow no\text{-step } cdcl_W\text{-cp } S \wedge no\text{-step } cdcl_W\text{-o } S$  (is ? $S' S \longleftrightarrow ?C S \wedge ?O S$ )
proof
  assume  $?C S \wedge ?O S$ 
  then show ? $S' S$ 

```

```

    by (auto simp: cdclW-s'.simps full1-def tranclp-unfold-begin)
next
assume n-s: ?S' S
have ?C S
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain S' where cdclW-cp S S'
  by auto
  then obtain T where full1 cdclW-cp S T
  using cdclW-cp-normalized-element-all-inv inv by (metis (no-types, lifting) full-unfold)
  then show False using n-s cdclW-s'.conflict' by blast
qed
moreover have ?O S
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain S' where cdclW-o S S'
  by auto
  then obtain T where full1 cdclW-cp S' T
  using cdclW-cp-normalized-element-all-inv inv
  by (meson cdclW-all-struct-inv-def n-s
    cdclW-stgy-cdclW-s'-connected' cdclW-then-exists-cdclW-stgy-step )
  then show False using n-s by (meson (cdclW-o S S') cdclW-all-struct-inv-def
    cdclW-stgy-cdclW-s'-connected' cdclW-then-exists-cdclW-stgy-step inv)
qed
ultimately show ?C S ∧ ?O S by auto
qed

lemma cdclW-s'-tranclp-cdclW:
  cdclW-s' S S' ⇒ cdclW++ S S'
proof (induct rule: cdclW-s'.induct)
  case conflict'
  then show ?case
  by (simp add: full1-def tranclp-cdclW-cp-tranclp-cdclW)
next
  case decide'
  then show ?case
  using cdclW-stgy.simps cdclW-stgy-tranclp-cdclW by (meson cdclW-o.simps)
next
  case (bj' Sa S'a S'') note a2 = this(1) and a1 = this(2) and n-s = this(3)
  obtain ss :: 'st ⇒ 'st ⇒ ('st ⇒ 'st ⇒ bool) ⇒ 'st where
    ∀ x0 x1 x2. (∃ v3. x2 x1 v3 ∧ x2** v3 x0) = (x2 x1 (ss x0 x1 x2) ∧ x2** (ss x0 x1 x2) x0)
  by moura
  then have f3: ∀ p s sa. ¬ p++ s sa ∨ p s (ss sa s p) ∧ p** (ss sa s p) sa
  by (metis (full-types) tranclpD)
  have cdclW-bj++ Sa S'a ∧ no-step cdclW-bj S'a
  using a2 by (simp add: full1-def)
  then have cdclW-bj Sa (ss S'a Sa cdclW-bj) ∧ cdclW-bj** (ss S'a Sa cdclW-bj) S'a
  using f3 by auto
  then show cdclW++ Sa S''
  using a1 n-s by (meson bj other rtranclp-cdclW-bj-full1-cdclp-cdclW-stgy
    rtranclp-cdclW-stgy-rtranclp-cdclW rtranclp-into-tranclp2)
qed

lemma tranclp-cdclW-s'-tranclp-cdclW:
  cdclW-s'++ S S' ⇒ cdclW++ S S'

```



```

apply (induct rule: tranclp.induct)
  using cdclW-s'-tranclp-cdclW apply blast
by (meson cdclW-s'-tranclp-cdclW tranclp-trans)

lemma rtranclp-cdclW-s'-rtranclp-cdclW:
  cdclW-s'^** S S'  $\implies$  cdclW^** S S'
  using rtranclp-unfold[of cdclW-s' S S'] tranclp-cdclW-s'-tranclp-cdclW[of S S'] by auto

lemma full-cdclW-stgy-iff-full-cdclW-s':
  assumes inv: cdclW-all-struct-inv S
  shows full cdclW-stgy S T  $\longleftrightarrow$  full cdclW-s' S T (is ?S  $\longleftrightarrow$  ?S')
proof
  assume ?S'
  then have cdclW^** S T
    using rtranclp-cdclW-s'-rtranclp-cdclW[of S T] unfolding full-def by blast
  then have inv': cdclW-all-struct-inv T
    using rtranclp-cdclW-all-struct-inv-inv inv by blast
  have cdclW-stgy^** S T
    using ⟨?S'⟩ unfolding full-def
    using cdclW-s'-is-rtranclp-cdclW-stgy rtranclp-mono[of cdclW-s' cdclW-stgy^**] by auto
  then show ?S
    using ⟨?S'⟩ inv' cdclW-stgy-cdclW-s'-connected' unfolding full-def by blast
next
  assume ?S
  then have inv-T:cdclW-all-struct-inv T
    by (metis assms full-def rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW)

consider
  (s') cdclW-s'^** S T
  | (st) S' where cdclW-s'^** S S' and cdclW-bj++ S' T and conflicting T  $\neq$  C-True
  using rtranclp-cdclW-stgy-connected-to-rtranclp-cdclW-s'[of S T] inv ⟨?S⟩
  unfolding full-def cdclW-all-struct-inv-def
  by blast
then show ?S'
  proof cases
    case s'
    then show ?thesis
      by (metis ⟨full cdclW-stgy S T⟩ inv-T cdclW-all-struct-inv-def cdclW-s'.simps
        cdclW-stgy.conflict' cdclW-then-exists-cdclW-stgy-step full-def
        n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  next
    case (st S')
    have full cdclW-cp T T
      using conflicting-clause-full-cdclW-cp st(3) by blast
    moreover
      have n-s: no-step cdclW-bj T
        by (metis ⟨full cdclW-stgy S T⟩ bj inv-T cdclW-all-struct-inv-def
          cdclW-then-exists-cdclW-stgy-step full-def)
      then have full1 cdclW-bj S' T
        using st(2) unfolding full1-def by blast
    moreover have no-step cdclW-cp S'
      using st(2) by (fastforce dest!: tranclpD simp: cdclW-cp.simps cdclW-bj.simps)
    ultimately have cdclW-s' S' T
      using cdclW-s'.bj'[of S' T T] by blast
    then have cdclW-s'^** S T

```

```

    using st(1) by auto
  moreover have no-step cdclW-s' T
    using inv-T by (metis ⟨full cdclW-cp T T⟩ ⟨full cdclW-stgy S T⟩ cdclW-all-struct-inv-def
      cdclW-then-exists-cdclW-stgy-step full-def n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  ultimately show ?thesis
    unfolding full-def by blast
qed
qed

```

**lemma** *conflict-step-cdcl<sub>W</sub>-stgy-step*:

```

  assumes
    conflict S T
    cdclW-all-struct-inv S
  shows ∃ T. cdclW-stgy S T
proof -
  obtain U where full cdclW-cp S U
    using cdclW-cp-normalized-element-all-inv assms by blast
  then have full1 cdclW-cp S U
    by (metis cdclW-cp.conflict' assms(1) full-unfold)
  then show ?thesis using cdclW-stgy.conflict' by blast
qed

```

**lemma** *decide-step-cdcl<sub>W</sub>-stgy-step*:

```

  assumes
    decide S T
    cdclW-all-struct-inv S
  shows ∃ T. cdclW-stgy S T
proof -
  obtain U where full cdclW-cp T U
    using cdclW-cp-normalized-element-all-inv by (meson assms(1) assms(2) cdclW-all-struct-inv-inv
      cdclW-cp-normalized-element-all-inv decide other)
  then show ?thesis
    by (metis assms cdclW-cp-normalized-element-all-inv cdclW-stgy.conflict' decide full-unfold
      other')
qed

```

**lemma** *rtrancp-cdcl<sub>W</sub>-cp-conflicting-C-Clause*:

```

  cdclW-cp** S T ⟹ conflicting S = C-Clause D ⟹ S = T
  using rtrancpD trancpD by fastforce

```

**inductive** *cdcl<sub>W</sub>-merge-cp* :: 'st ⇒ 'st ⇒ bool **where**

```

  conflict'[intro]: conflict S T ⟹ full cdclW-bj T U ⟹ cdclW-merge-cp S U |
  propagate'[intro]: propagate++ S S' ⟹ cdclW-merge-cp S S'

```

**lemma** *cdcl<sub>W</sub>-merge-restart-cases*[consumes 1, case-names conflict propagate]:

```

  assumes
    cdclW-merge-cp S U and
    ∧ T. conflict S T ⟹ full cdclW-bj T U ⟹ P and
    propagate++ S U ⟹ P
  shows P
  using assms unfolding cdclW-merge-cp.simps by auto

```

**lemma** *cdcl<sub>W</sub>-merge-cp-trancp-cdcl<sub>W</sub>-merge*:

```

  cdclW-merge-cp S T ⟹ cdclW-merge++ S T
  apply (induction rule: cdclW-merge-cp.induct)

```

```

    using cdclW-merge.simps apply auto[1]
using tranclp-mono[of propagate cdclW-merge] fw-propagate by blast

lemma rtranclp-cdclW-merge-cp-rtranclp-cdclW:
  cdclW-merge-cp** S T  $\implies$  cdclW** S T
apply (induction rule: rtranclp-induct)
apply simp
unfolding cdclW-merge-cp.simps by (meson cdclW-merge-restart-cdclW fw-r-conflict
  rtranclp-propagate-is-rtranclp-cdclW rtranclp-trans tranclp-into-rtranclp)

lemma full1-cdclW-bj-no-step-cdclW-bj:
  full1 cdclW-bj S T  $\implies$  no-step cdclW-cp S
by (metis rtranclp-unfold cdclW-cp-conflicting-not-empty conflicting-clause.exhaust full1-def
  rtranclp-cdclW-merge-restart-no-step-cdclW-bj tranclpD)

inductive cdclW-s'-without-decide where
  conflict'-without-decide[intro]: full1 cdclW-cp S S'  $\implies$  cdclW-s'-without-decide S S' |
  bj'-without-decide[intro]: full1 cdclW-bj S S'  $\implies$  no-step cdclW-cp S  $\implies$  full cdclW-cp S' S''
     $\implies$  cdclW-s'-without-decide S S''

lemma rtranclp-cdclW-s'-without-decide-rtranclp-cdclW:
  cdclW-s'-without-decide** S T  $\implies$  cdclW** S T
apply (induction rule: rtranclp-induct)
apply simp
by (meson cdclW-s'.simps cdclW-s'-tranclp-cdclW cdclW-s'-without-decide.simps
  rtranclp-tranclp-tranclp tranclp-into-rtranclp)

lemma rtranclp-cdclW-s'-without-decide-rtranclp-cdclW-s':
  cdclW-s'-without-decide** S T  $\implies$  cdclW-s'*** S T
proof (induction rule: rtranclp-induct)
  case base
  then show ?case by simp
next
  case (step y z) note a2 = this(2) and a1 = this(3)
  have cdclW-s' y z
    using a2 by (metis (no-types) bj' cdclW-s'.conflict' cdclW-s'-without-decide.cases)
  then show cdclW-s'*** S z
    using a1 by (meson r-into-rtranclp rtranclp-trans)
qed

lemma rtranclp-cdclW-merge-cp-is-rtranclp-cdclW-s'-without-decide:
  assumes
    cdclW-merge-cp** S V
    conflicting S = C-True
  shows
    (cdclW-s'-without-decide** S V)
     $\vee$  ( $\exists T$ . cdclW-s'-without-decide** S T  $\wedge$  propagate++ T V)
     $\vee$  ( $\exists T U$ . cdclW-s'-without-decide** S T  $\wedge$  full1 cdclW-bj T U  $\wedge$  propagate** U V)
  using assms
proof (induction rule: rtranclp-induct)
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and cp = this(2) and IH = this(3)[OF this(4)]
  from cp show ?case

```

```

proof (cases rule: cdclW-merge-restart-cases)
  case propagate
  then show ?thesis using IH by (meson rtrancpl-trancpl-trancpl trancpl-into-rtrancpl)
next
  case (conflict U') note confl = this(1) and bj = this(2)
  have full1-U-U': full1 cdclW-cp U U'
    by (simp add: conflict-is-full1-cdclW-cp local.conflict(1))
  consider
    (s') cdclW-s'-without-decide** S U
  | (propa) T' where cdclW-s'-without-decide** S T' and propagate++ T' U
  | (bj-prop) T' T'' where
    cdclW-s'-without-decide** S T' and
    full1 cdclW-bj T' T'' and
    propagate** T'' U
  using IH by blast
then show ?thesis
proof cases
  case s'
  have cdclW-s'-without-decide U U'
    using full1-U-U' conflict'-without-decide by blast
  then have cdclW-s'-without-decide** S U'
    using ⟨cdclW-s'-without-decide** S U⟩ by auto
  moreover have U' = V ∨ full1 cdclW-bj U' V
    using bj by (meson full-unfold)
  ultimately show ?thesis by blast
next
  case propa note s' = this(1) and T'-U = this(2)
  have full1 cdclW-cp T' U'
    using rtrancpl-mono[of propagate cdclW-cp] T'-U cdclW-cp.propagate' full1-U-U'
    rtrancpl-full1I[of cdclW-cp T'] by (metis (full-types) predicate2D predicate2I
      trancpl-into-rtrancpl)
  have cdclW-s'-without-decide** S U'
    using ⟨full1 cdclW-cp T' U'⟩ conflict'-without-decide s' by force
  have full1 cdclW-bj U' V ∨ V = U'
    by (metis (lifting) full-unfold local.bj)
  then show ?thesis
    using ⟨cdclW-s'-without-decide** S U'⟩ by blast
next
  case bj-prop note s' = this(1) and bj-T' = this(2) and T''-U = this(3)
  have no-step cdclW-cp T'
    using bj-T' full1-cdclW-bj-no-step-cdclW-bj by blast
  moreover have full1 cdclW-cp T'' U'
    using rtrancpl-mono[of propagate cdclW-cp] T''-U cdclW-cp.propagate' full1-U-U'
    rtrancpl-full1I[of cdclW-cp T''] by blast
  ultimately have cdclW-s'-without-decide T' U'
    using bj'-without-decide[of T' T'' U'] bj-T' by (simp add: full-unfold)
  then have cdclW-s'-without-decide** S U'
    using s' rtrancpl.intros(2)[of - S T' U'] by blast
  then show ?thesis
    by (metis full-unfold local.bj rtrancpl.rtrancpl-refl)
qed
qed
qed

```

**lemma** *rtrancp-cdcl<sub>W</sub>-s'-without-decide-is-rtrancp-cdcl<sub>W</sub>-merge-cp*:

**assumes**

*cdcl<sub>W</sub>-s'-without-decide*\*\* *S V* **and**

*confl*: *conflicting S = C-True*

**shows**

(*cdcl<sub>W</sub>-merge-cp*\*\* *S V*  $\wedge$  *conflicting V = C-True*)

$\vee$  (*cdcl<sub>W</sub>-merge-cp*\*\* *S V*  $\wedge$  *conflicting V*  $\neq$  *C-True*  $\wedge$  *no-step cdcl<sub>W</sub>-cp V*  $\wedge$  *no-step cdcl<sub>W</sub>-bj V*)

$\vee$  ( $\exists T. \text{cdcl}_W\text{-merge-cp}^{**} S T \wedge \text{conflict } T V$ )

**using** *assms(1)*

**proof** (*induction*)

**case** *base*

**then show** *?case* **using** *confl* **by** *auto*

**next**

**case** (*step U V*) **note** *st = this(1)* **and** *s = this(2)* **and** *IH = this(3)*

**from** *s* **show** *?case*

**proof** (*cases rule: cdcl<sub>W</sub>-s'-without-decide.cases*)

**case** *conflict'-without-decide*

**then have** *rt: cdcl<sub>W</sub>-cp<sup>++</sup> U V* **unfolding** *full1-def* **by** *fast*

**then have** *conflicting U = C-True*

**using** *trancp-cdcl<sub>W</sub>-cp-propagate-with-conflict-or-not[of U V]*

**conflict** **by** (*auto dest!: trancpD simp: rtrancp-unfold*)

**then have** *cdcl<sub>W</sub>-merge-cp<sup>\*\*</sup> S U* **using** *IH* **by** *auto*

**consider**

(*propa*) *propagate<sup>++</sup> U V*

| (*confl'*) *conflict U V*

| (*propa-confl'*) *U'* **where** *propagate<sup>++</sup> U U' conflict U' V*

**using** *trancp-cdcl<sub>W</sub>-cp-propagate-with-conflict-or-not[OF rt]* **unfolding** *rtrancp-unfold*

**by** *fastforce*

**then show** *?thesis*

**proof** *cases*

**case** *propa*

**then have** *cdcl<sub>W</sub>-merge-cp U V*

**by** *auto*

**moreover have** *conflicting V = C-True*

**using** *propa* **unfolding** *trancp-unfold-end* **by** *auto*

**ultimately show** *?thesis* **using**  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  **by** *force*

**next**

**case** *confl'*

**then show** *?thesis* **using**  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  **by** *auto*

**next**

**case** *propa-confl'* **note** *propa = this(1)* **and** *confl' = this(2)*

**then have** *cdcl<sub>W</sub>-merge-cp U U'* **by** *auto*

**then have** *cdcl<sub>W</sub>-merge-cp<sup>\*\*</sup> S U'* **using**  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  **by** *auto*

**then show** *?thesis* **using**  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  *confl'* **by** *auto*

**qed**

**next**

**case** (*bj'-without-decide U'*) **note** *full-bj = this(1)* **and** *cp = this(3)*

**then have** *conflicting U*  $\neq$  *C-True*

**using** *full-bj* **unfolding** *full1-def* **by** (*fastforce dest!: trancpD simp: cdcl<sub>W</sub>-bj.simps*)

**with** *IH* **obtain** *T* **where**

*S-T: cdcl<sub>W</sub>-merge-cp<sup>\*\*</sup> S T* **and** *T-U: conflict T U*

**using** *full-bj* **unfolding** *full1-def* **by** (*blast dest: trancpD*)

**then have** *cdcl<sub>W</sub>-merge-cp T U'*

**using** *cdcl<sub>W</sub>-merge-cp.conflict'[of T U U']* *full-bj* **by** (*simp add: full-unfold*)

**then have** *S-U': cdcl<sub>W</sub>-merge-cp<sup>\*\*</sup> S U'* **using** *S-T* **by** *auto*

```

consider
  (n-s)  $U' = V$ 
  | (propa)  $\text{propagate}^{++} U' V$ 
  | (confl')  $\text{conflict } U' V$ 
  | (propa-confl')  $U''$  where  $\text{propagate}^{++} U' U'' \text{ conflict } U'' V$ 
using trancpl-cdclW-cp-propagate-with-conflict-or-not cp
unfolding rtrancpl-unfold full-def by metis
then show ?thesis
proof cases
  case propa
    then have cdclW-merge-cp U' V by auto
    moreover have conflicting V = C-True
      using propa unfolding trancpl-unfold-end by auto
    ultimately show ?thesis using S-U' by force
  next
    case confl'
      then show ?thesis using S-U' by auto
  next
    case propa-confl' note propa = this(1) and confl = this(2)
    have cdclW-merge-cp U' U'' using propa by auto
    then show ?thesis using S-U' confl by (meson rtrancpl.rtrancpl-into-rtrancpl)
  next
    case n-s
      then show ?thesis
        using S-U' apply (cases conflicting V = C-True)
        using full-bj apply simp
        by (metis cp full-def full-unfold full-bj)
  qed
qed
qed

```

**lemma** *no-step-cdcl<sub>W</sub>-s'-no-ste-cdcl<sub>W</sub>-merge-cp:*

```

assumes
  cdclW-all-struct-inv S
  conflicting S = C-True
  no-step cdclW-s' S
shows no-step cdclW-merge-cp S
using assms apply (auto simp: cdclW-s'.simps cdclW-merge-cp.simps)
using conflict-is-full1-cdclW-cp apply blast
using cdclW-cp-normalized-element-all-inv cdclW-cp.propagate' by (metis cdclW-cp.propagate' full-unfold trancplD)

```

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
  conft: conflicting S = C-True and
  inv: cdclW-M-level-inv S and
  n-s: no-step cdclW-merge-cp S
shows no-step cdclW-s'-without-decide S
proof (rule ccontr)
  assume  $\neg$  no-step cdclW-s'-without-decide S
  then obtain T where
    cdclW: cdclW-s'-without-decide S T
    by auto
  then have inv-T: cdclW-M-level-inv T

```

```

using rtranclp-cdclW-s'-without-decide-rtranclp-cdclW[of S T]
rtranclp-cdclW-consistent-inv inv by blast
from cdclW show False
proof cases
  case conflict'-without-decide
  have no-step propagate S
    using n-s by blast
  then have conflict S T
    using local.conflict' tranclp-cdclW-cp-propagate-with-conflict-or-not[of S T]
    unfolding full1-def by (metis full1-def local.conflict'-without-decide rtranclp-unfold
      tranclp-unfold-begin)
  moreover
    then obtain T' where full cdclW-bj T T'
    using cdclW-bj-exists-normal-form inv-T by blast
  ultimately show False using cdclW-merge-cp.conflict' n-s by meson
next
  case (bj'-without-decide S')
  then show ?thesis
    using confl unfolding full1-def by (fastforce simp: cdclW-bj.simps dest: tranclpD)
qed
qed

```

**lemma** *conflicting-true-no-step-s'-without-decide-no-step-cdcl<sub>W</sub>-merge-cp:*

```

assumes
  inv: cdclW-all-struct-inv S and
  n-s: no-step cdclW-s'-without-decide S
shows no-step cdclW-merge-cp S
proof (rule ccontr)
assume  $\neg$  ?thesis
then obtain T where cdclW-merge-cp S T
  by auto
then show False
proof cases
  case (conflict' S')
  then show False using n-s conflict'-without-decide conflict-is-full1-cdclW-cp by blast
next
  case propagate'
  moreover
    have cdclW-all-struct-inv T
      using inv by (meson local.propagate' rtranclp-cdclW-all-struct-inv-inv
        rtranclp-propagate-is-rtranclp-cdclW tranclp-into-rtranclp)
    then obtain U where full cdclW-cp T U
      using cdclW-cp-normalized-element-all-inv by auto
    ultimately have full1 cdclW-cp S U
      using tranclp-full-full1I[of cdclW-cp S T U] cdclW-cp.propagate'
      tranclp-mono[of propagate cdclW-cp] by blast
    then show False using conflict'-without-decide n-s by blast
qed
qed

```

**lemma** *no-step-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-cp:*

```

no-step cdclW-merge-cp S  $\implies$  cdclW-M-level-inv S  $\implies$  no-step cdclW-cp S
using cdclW-bj-exists-normal-form cdclW-consistent-inv[OF cdclW.conflict, of S]
by (metis cdclW-cp.cases cdclW-merge-cp.simps tranclp.intros(1))

```

**lemma** *conflicting-not-true-rtrancp-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-bj*:  
**assumes**  
*conflicting*  $S = C\text{-True}$  **and**  
*cdcl<sub>W</sub>-merge-cp*\*\*  $S\ T$   
**shows** *no-step cdcl<sub>W</sub>-bj*  $T$   
**using** *assms*(2,1) **by** (*induction*)  
*(fastforce simp: cdcl<sub>W</sub>-merge-cp.simps full-def trancp-unfold-end cdcl<sub>W</sub>-bj.simps)*+

**lemma** *conflicting-true-full-cdcl<sub>W</sub>-merge-cp-iff-full-cdcl<sub>W</sub>-s'-without-decode*:  
**assumes**  
*confl*: *conflicting*  $S = C\text{-True}$  **and**  
*inv*: *cdcl<sub>W</sub>-all-struct-inv*  $S$   
**shows**  
*full cdcl<sub>W</sub>-merge-cp*  $S\ V \longleftrightarrow \text{full cdcl}_W\text{-s'-without-decode } S\ V$  (**is**  $?fw \longleftrightarrow ?s'$ )

**proof**  
**assume**  $?fw$   
**then have** *st*: *cdcl<sub>W</sub>-merge-cp*\*\*  $S\ V$  **and** *n-s*: *no-step cdcl<sub>W</sub>-merge-cp*  $V$   
**unfolding** *full-def* **by** *blast* +  
**have** *inv-V*: *cdcl<sub>W</sub>-all-struct-inv*  $V$   
**using** *rtrancp-cdcl<sub>W</sub>-merge-cp-rtrancp-cdcl<sub>W</sub>[of S V]*  $\langle ?fw \rangle$  **unfolding** *full-def*  
**by** (*simp add: inv rtrancp-cdcl<sub>W</sub>-all-struct-inv-inv*)  
**consider**  
 $(s')$  *cdcl<sub>W</sub>-s'-without-decode*\*\*  $S\ V$   
 $|$  (*propa*)  $T$  **where** *cdcl<sub>W</sub>-s'-without-decode*\*\*  $S\ T$  **and** *propagate*<sup>++</sup>  $T\ V$   
 $|$  (*bj*)  $T\ U$  **where** *cdcl<sub>W</sub>-s'-without-decode*\*\*  $S\ T$  **and** *full1 cdcl<sub>W</sub>-bj*  $T\ U$  **and** *propagate*\*\*  $U\ V$   
**using** *rtrancp-cdcl<sub>W</sub>-merge-cp-is-rtrancp-cdcl<sub>W</sub>-s'-without-decode confl st n-s* **by** *metis*  
**then have** *cdcl<sub>W</sub>-s'-without-decode*\*\*  $S\ V$   
**proof cases**  
**case**  $s'$   
**then show** *?thesis* .  
**next**  
**case** *propa* **note**  $s' = \text{this}(1)$  **and** *propa* = *this*(2)  
**have** *no-step cdcl<sub>W</sub>-cp*  $V$   
**using** *no-step-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-cp n-s inv-V*  
**unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *blast*  
**then have** *full1 cdcl<sub>W</sub>-cp*  $T\ V$   
**using** *propa trancp-mono[of propagate cdcl<sub>W</sub>-cp] cdcl<sub>W</sub>-cp.propagate'* **unfolding** *full1-def*  
**by** *blast*  
**then have** *cdcl<sub>W</sub>-s'-without-decode*  $T\ V$   
**using** *conflict'-without-decode* **by** *blast*  
**then show** *?thesis* **using**  $s'$  **by** *auto*  
**next**  
**case** *bj* **note**  $s' = \text{this}(1)$  **and** *bj* = *this*(2) **and** *propa* = *this*(3)  
**have** *no-step cdcl<sub>W</sub>-cp*  $V$   
**using** *no-step-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-cp n-s inv-V*  
**unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *blast*  
**then have** *full cdcl<sub>W</sub>-cp*  $U\ V$   
**using** *propa rtrancp-mono[of propagate cdcl<sub>W</sub>-cp] cdcl<sub>W</sub>-cp.propagate'* **unfolding** *full-def*  
**by** *blast*  
**moreover have** *no-step cdcl<sub>W</sub>-cp*  $T$   
**using** *bj unfolding full1-def* **by** (*fastforce dest!: trancpD simp:cdcl<sub>W</sub>-bj.simps*)  
**ultimately have** *cdcl<sub>W</sub>-s'-without-decode*  $T\ V$   
**using** *bj'-without-decode[of T U V]* *bj* **by** *blast*  
**then show** *?thesis* **using**  $s'$  **by** *auto*  
**qed**



```

moreover have no-step cdclW-s'-without-decide V
proof (cases conflicting V = C-True)
  case False
  { fix ss :: 'st
    have ff1:  $\forall s \text{ sa. } \neg \text{cdcl}_W\text{-s'} \text{ s sa} \vee \text{full1 cdcl}_W\text{-cp s sa}$ 
       $\vee (\exists sb. \text{decide s sb} \wedge \text{no-step cdcl}_W\text{-cp s} \wedge \text{full cdcl}_W\text{-cp sb sa})$ 
       $\vee (\exists sb. \text{full1 cdcl}_W\text{-bj s sb} \wedge \text{no-step cdcl}_W\text{-cp s} \wedge \text{full cdcl}_W\text{-cp sb sa})$ 
      by (metis cdclW-s'.cases)
    have ff2:  $(\forall p \text{ s sa. } \neg \text{full1 p (s::'st) sa} \vee p^{++} \text{ s sa} \wedge \text{no-step p sa})$ 
       $\wedge (\forall p \text{ s sa. } (\neg p^{++} \text{ (s::'st) sa} \vee (\exists s. p \text{ sa s})) \vee \text{full1 p s sa})$ 
      by (meson full1-def)
    obtain ssa :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
      ff3:  $\forall p \text{ s sa. } \neg p^{++} \text{ s sa} \vee p \text{ s (ssa p s sa)} \wedge p^{**} \text{ (ssa p s sa) sa}$ 
      by (metis (no-types) tranclpD)
    then have a3:  $\neg \text{cdcl}_W\text{-cp}^{++} \text{ V ss}$ 
      using False by (metis conflicting-clause-full-cdclW-cp full-def)
    have  $\bigwedge s. \neg \text{cdcl}_W\text{-bj}^{++} \text{ V s}$ 
      using ff3 False by (metis confl st
        conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj)
    then have  $\neg \text{cdcl}_W\text{-s'-without-decide V ss}$ 
      using ff1 a3 ff2 by (metis cdclW-s'-without-decide.cases)
  }
then show ?thesis
  by fastforce
next
  case True
  then show ?thesis
    using conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide n-s inv-V
    unfolding cdclW-all-struct-inv-def by blast
  qed
ultimately show ?s' unfolding full-def by blast
next
assume s': ?s'
then have st: cdclW-s'-without-decide** S V and n-s: no-step cdclW-s'-without-decide V
  unfolding full-def by auto
then have cdclW** S V
  using rtranclp-cdclW-s'-without-decide-rtranclp-cdclW st by blast
then have inv-V: cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
then have n-s-cp-V: no-step cdclW-cp V
  using cdclW-cp-normalized-element-all-inv[of V] full-fullI[of cdclW-cp V] n-s
  conflict'-without-decide conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp
  no-step-cdclW-merge-cp-no-step-cdclW-cp
  unfolding cdclW-all-struct-inv-def by presburger
have n-s-bj: no-step cdclW-bj V
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain W where W: cdclW-bj V W by blast
  have cdclW-all-struct-inv W
    using W cdclW.simps cdclW-all-struct-inv-inv inv-V by blast
  then obtain W' where full1 cdclW-bj V W'
    using cdclW-bj-exists-normal-form[of W] full-fullI[of cdclW-bj V W] W
    unfolding cdclW-all-struct-inv-def
    by blast
moreover
  then have cdclW^{++} V W'

```

```

    using trancpl-mono[of cdclW-bj cdclW] cdclW.other cdclW-o.bj unfolding full1-def by blast
  then have cdclW-all-struct-inv W'
    by (meson inv-V rtrancpl-cdclW-all-struct-inv-inv trancpl-into-rtrancpl)
  then obtain X where full cdclW-cp W' X
    using cdclW-cp-normalized-element-all-inv by blast
  ultimately show False
    using bj'-without-decide n-s-cp-V n-s by blast
qed
from s' consider
  (cp-true) cdclW-merge-cp** S V and conflicting V = C-True
| (cp-false) cdclW-merge-cp** S V and conflicting V ≠ C-True and no-step cdclW-cp V and
  no-step cdclW-bj V
| (cp-conf) T where cdclW-merge-cp** S T conflict T V
  using rtrancpl-cdclW-s'-without-decide-is-rtrancpl-cdclW-merge-cp[of S V] confl
  unfolding full-def by blast
then have cdclW-merge-cp** S V
  proof cases
    case cp-conf note S-T = this(1) and conf-V = this(2)
    have full cdclW-bj V V
      using conf-V n-s-bj unfolding full-def by fast
    then have cdclW-merge-cp T V
      using cdclW-merge-cp.conflict' conf-V by auto
    then show ?thesis using S-T by auto
  qed fast+
moreover
  then have cdclW** S V using rtrancpl-cdclW-merge-cp-rtrancpl-cdclW by blast
  then have cdclW-all-struct-inv V
    using inv rtrancpl-cdclW-all-struct-inv-inv by blast
  then have no-step cdclW-merge-cp V
    using conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp s'
    unfolding full-def by blast
  ultimately show ?fw unfolding full-def by auto
qed

lemma conflicting-true-full1-cdclW-merge-cp-iff-full1-cdclW-s'-without-decode:
  assumes
    confl: conflicting S = C-True and
    inv: cdclW-all-struct-inv S
  shows
    full1 cdclW-merge-cp S V ⟷ full1 cdclW-s'-without-decide S V
proof –
  have full cdclW-merge-cp S V = full cdclW-s'-without-decide S V
    using confl conflicting-true-full-cdclW-merge-cp-iff-full-cdclW-s'-without-decode inv
    by blast
  then show ?thesis unfolding full-unfold full1-def
    by (metis (mono-tags) trancpl-unfold-begin)
qed

lemma conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode:
  assumes
    fw: full1 cdclW-merge-cp S V and
    inv: cdclW-all-struct-inv S
  shows
    full1 cdclW-s'-without-decide S V
proof –

```

**have** *conflicting*  $S = C\text{-True}$   
**using** *fw unfolding full1-def* **by** (*auto dest!:* *trancplD simp: cdcl<sub>W</sub>-merge-cp.simps*)  
**then show** *?thesis*  
**using** *conflicting-true-full1-cdcl<sub>W</sub>-merge-cp-iff-full1-cdcl<sub>W</sub>-s'-without-decode fw inv* **by** *blast*  
**qed**

**inductive** *cdcl<sub>W</sub>-merge-stgy* **where**

*fw-s-cp[intro]: full1 cdcl<sub>W</sub>-merge-cp S T  $\implies$  cdcl<sub>W</sub>-merge-stgy S T* |

*fw-s-decide[intro]: decide S T  $\implies$  no-step cdcl<sub>W</sub>-merge-cp S  $\implies$  full cdcl<sub>W</sub>-merge-cp T U*  
 $\implies$  *cdcl<sub>W</sub>-merge-stgy S U*

**lemma** *cdcl<sub>W</sub>-merge-stgy-trancpl-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*

**proof** –

{ **fix** *S T*

**assume** *full1 cdcl<sub>W</sub>-merge-cp S T*

**then have** *cdcl<sub>W</sub>-merge<sup>++</sup> S T*

**using** *trancpl-mono[of cdcl<sub>W</sub>-merge-cp cdcl<sub>W</sub>-merge<sup>++</sup>] cdcl<sub>W</sub>-merge-cp-trancpl-cdcl<sub>W</sub>-merge*

**unfolding** *full1-def*

**by** *auto*

} **note** *full1-cdcl<sub>W</sub>-merge-cp-cdcl<sub>W</sub>-merge = this*

**show** *?thesis*

**using** *fw*

**apply** (*induction rule: cdcl<sub>W</sub>-merge-stgy.induct*)

**using** *full1-cdcl<sub>W</sub>-merge-cp-cdcl<sub>W</sub>-merge* **apply** *simp*

**unfolding** *full-unfold* **by** (*auto dest!:* *full1-cdcl<sub>W</sub>-merge-cp-cdcl<sub>W</sub>-merge fw-decide*)

**qed**

**lemma** *rtrancpl-cdcl<sub>W</sub>-merge-stgy-rtrancpl-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*

**using** *fw cdcl<sub>W</sub>-merge-stgy-trancpl-cdcl<sub>W</sub>-merge* *rtrancpl-mono[of cdcl<sub>W</sub>-merge-stgy cdcl<sub>W</sub>-merge<sup>++</sup>]*

**unfolding** *trancpl-rtrancpl-rtrancpl* **by** *blast*

**lemma** *cdcl<sub>W</sub>-merge-stgy-rtrancpl-cdcl<sub>W</sub>:*

*cdcl<sub>W</sub>-merge-stgy S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*

**apply** (*induction rule: cdcl<sub>W</sub>-merge-stgy.induct*)

**using** *rtrancpl-cdcl<sub>W</sub>-merge-cp-rtrancpl-cdcl<sub>W</sub>* **unfolding** *full1-def*

**apply** (*simp add: trancpl-into-rtrancpl*)

**using** *rtrancpl-cdcl<sub>W</sub>-merge-cp-rtrancpl-cdcl<sub>W</sub> cdcl<sub>W</sub>-o.decide cdcl<sub>W</sub>-other* **unfolding** *full-def*

**by** (*meson r-into-rtrancpl rtrancpl-trans*)

**lemma** *rtrancpl-cdcl<sub>W</sub>-merge-stgy-rtrancpl-cdcl<sub>W</sub>:*

*cdcl<sub>W</sub>-merge-stgy<sup>\*\*</sup> S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*

**using** *rtrancpl-mono[of cdcl<sub>W</sub>-merge-stgy cdcl<sub>W</sub><sup>\*\*</sup>]* *cdcl<sub>W</sub>-merge-stgy-rtrancpl-cdcl<sub>W</sub>* **by** *auto*

**inductive** *cdcl<sub>W</sub>-s'-w* :: *'st  $\Rightarrow$  'st  $\Rightarrow$  bool* **where**

*conflict': full1 cdcl<sub>W</sub>-s'-without-decide S S'  $\implies$  cdcl<sub>W</sub>-s'-w S S'* |

*decide': decide S S'  $\implies$  no-step cdcl<sub>W</sub>-s'-without-decide S  $\implies$  full cdcl<sub>W</sub>-s'-without-decide S' S''*  
 $\implies$  *cdcl<sub>W</sub>-s'-w S S''*

**lemma** *cdcl<sub>W</sub>-s'-w-rtrancpl-cdcl<sub>W</sub>:*

*cdcl<sub>W</sub>-s'-w S T  $\implies$  cdcl<sub>W</sub><sup>\*\*</sup> S T*

**apply** (*induction rule: cdcl<sub>W</sub>-s'-w.induct*)

```

    using rtrancpl-cdclW-s'-without-decide-rtrancpl-cdclW unfolding full1-def
    apply (simp add: trancpl-into-rtrancpl)
using rtrancpl-cdclW-s'-without-decide-rtrancpl-cdclW unfolding full-def
by (meson decide other rtrancpl-into-trancpl2 trancpl-into-rtrancpl)

lemma rtrancpl-cdclW-s'-w-rtrancpl-cdclW:
  cdclW-s'-w** S T  $\implies$  cdclW** S T
  using rtrancpl-mono[of cdclW-s'-w cdclW**] cdclW-s'-w-rtrancpl-cdclW by auto

lemma no-step-cdclW-cp-no-step-cdclW-s'-without-decide:
  assumes no-step cdclW-cp S and conflicting S = C-True and inv: cdclW-M-level-inv S
  shows no-step cdclW-s'-without-decide S
  by (metis assms cdclW-cp.conflict' cdclW-cp.propagate' cdclW-merge-restart-cases trancplD
      conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide)

lemma no-step-cdclW-cp-no-step-cdclW-merge-restart:
  assumes no-step cdclW-cp S and conflicting S = C-True
  shows no-step cdclW-merge-cp S
  by (metis assms(1) cdclW-cp.conflict' cdclW-cp.propagate' cdclW-merge-restart-cases trancplD)
lemma after-cdclW-s'-without-decide-no-step-cdclW-cp:
  assumes cdclW-s'-without-decide S T
  shows no-step cdclW-cp T
  using assms by (induction rule: cdclW-s'-without-decide.induct) (auto simp: full1-def full-def)

lemma no-step-cdclW-s'-without-decide-no-step-cdclW-cp:
  cdclW-all-struct-inv S  $\implies$  no-step cdclW-s'-without-decide S  $\implies$  no-step cdclW-cp S
  by (simp add: conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp
      no-step-cdclW-merge-cp-no-step-cdclW-cp cdclW-all-struct-inv-def)

lemma after-cdclW-s'-w-no-step-cdclW-cp:
  assumes cdclW-s'-w S T and cdclW-all-struct-inv S
  shows no-step cdclW-cp T
  using assms
proof (induction rule: cdclW-s'-w.induct)
  case conflict'
  then show ?case
    by (auto simp: full1-def trancpl-unfold-end after-cdclW-s'-without-decide-no-step-cdclW-cp)
next
  case (decide' S T U)
  moreover
    then have cdclW** S U
      using rtrancpl-cdclW-s'-without-decide-rtrancpl-cdclW[of T U] cdclW.other[of S T]
      cdclW-o.decide unfolding full-def by auto
    then have cdclW-all-struct-inv U
      using decide'.prems rtrancpl-cdclW-all-struct-inv-inv by blast
  ultimately show ?case
    using no-step-cdclW-s'-without-decide-no-step-cdclW-cp unfolding full-def by blast
qed

lemma rtrancpl-cdclW-s'-w-no-step-cdclW-cp-or-eq:
  assumes cdclW-s'-w** S T and cdclW-all-struct-inv S
  shows S = T  $\vee$  no-step cdclW-cp T
  using assms
proof (induction rule: rtrancpl-induct)
  case base

```

```

then show ?case by simp
next
case (step T U)
moreover have cdclW-all-struct-inv T
  using rtrancpl-cdclW-s'-w-rtrancpl-cdclW[of S U] assms(2) rtrancpl-cdclW-all-struct-inv-inv
  rtrancpl-cdclW-s'-w-rtrancpl-cdclW step.hyps(1) by blast
ultimately show ?case using after-cdclW-s'-w-no-step-cdclW-cp by fast
qed

lemma rtrancpl-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq:
  assumes cdclW-merge-stgy** S T and inv: cdclW-all-struct-inv S
  shows S = T ∨ no-step cdclW-cp T
  using assms
proof (induction rule: rtrancpl-induct)
  case base
  then show ?case by simp
next
  case (step T U)
  moreover have cdclW-all-struct-inv T
    using rtrancpl-cdclW-merge-stgy-rtrancpl-cdclW[of S U] assms(2) rtrancpl-cdclW-all-struct-inv-inv
    rtrancpl-cdclW-s'-w-rtrancpl-cdclW step.hyps(1)
    by (meson rtrancpl-cdclW-merge-stgy-rtrancpl-cdclW)
  ultimately show ?case
    using after-cdclW-s'-w-no-step-cdclW-cp inv unfolding cdclW-all-struct-inv-def
    by (metis cdclW-all-struct-inv-def cdclW-merge-stgy.simps full1-def full-def
      no-step-cdclW-merge-cp-no-step-cdclW-cp rtrancpl-cdclW-all-struct-inv-inv
      rtrancpl-cdclW-merge-stgy-rtrancpl-cdclW trancpl.intros(1) trancpl-into-rtrancpl)
qed

lemma no-step-cdclW-s'-without-decide-no-step-cdclW-bj:
  assumes no-step cdclW-s'-without-decide S and inv: cdclW-all-struct-inv S
  shows no-step cdclW-bj S
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain T where S-T: cdclW-bj S T
    by auto
  have cdclW-all-struct-inv T
    using S-T cdclW-all-struct-inv-inv inv other by blast
  then obtain T' where full1 cdclW-bj S T'
    using cdclW-bj-exists-normal-form[of T] full-fullI S-T unfolding cdclW-all-struct-inv-def
    by metis
  moreover
  then have cdclW** S T'
    using rtrancpl-mono[of cdclW-bj cdclW] cdclW.other cdclW-o.bj trancpl-into-rtrancpl[of cdclW-bj]
    unfolding full1-def by (metis (full-types) predicate2D predicate2I)
  then have cdclW-all-struct-inv T'
    using inv rtrancpl-cdclW-all-struct-inv-inv by blast
  then obtain U where full cdclW-cp T' U
    using cdclW-cp-normalized-element-all-inv by blast
  moreover have no-step cdclW-cp S
    using S-T by (auto simp: cdclW-bj.simps)
  ultimately show False
    using assms cdclW-s'-without-decide.intros(2)[of S T' U] by fast
qed

```

**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**  $no-step\ cdcl_W-bj\ T$   
**using** *assms apply induction*  
**using**  $rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W\ rtrancpl-cdcl_W-all-struct-inv-inv$   
 $no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj$  **unfolding** *full1-def*  
**apply** (*meson trancpl-into-rtrancpl*)  
**using**  $rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W\ rtrancpl-cdcl_W-all-struct-inv-inv$   
 $no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj$  **unfolding** *full-def*  
**by** (*meson cdcl\_W-merge-restart-cdcl\_W fw-r-decide*)

**lemma**  $rtrancpl-cdcl_W-s'-w-no-step-cdcl_W-bj-or-eq$ :  
**assumes**  $cdcl_W-s'-w^{**}\ S\ T$  **and**  $cdcl_W-all-struct-inv\ S$   
**shows**  $S = T \vee no-step\ cdcl_W-bj\ T$   
**using** *assms apply induction*  
**apply** *simp*  
**using**  $rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W\ rtrancpl-cdcl_W-all-struct-inv-inv$   
 $cdcl_W-s'-w-no-step-cdcl_W-bj$  **by** *meson*

**lemma**  $rtrancpl-cdcl_W-s'-no-step-cdcl_W-s'-without-decide-decomp-into-cdcl_W-merge$ :  
**assumes**  
 $cdcl_W-s'^{**}\ R\ V$  **and**  
 $conflicting\ R = C-True$  **and**  
 $inv: cdcl_W-all-struct-inv\ R$   
**shows** ( $cdcl_W-merge-stgy^{**}\ R\ V \wedge conflicting\ V = C-True$ )  
 $\vee (cdcl_W-merge-stgy^{**}\ R\ V \wedge conflicting\ V \neq C-True \wedge no-step\ cdcl_W-bj\ V)$   
 $\vee (\exists\ S\ T\ U. cdcl_W-merge-stgy^{**}\ R\ S \wedge no-step\ cdcl_W-merge-cp\ S \wedge decide\ S\ T$   
 $\wedge cdcl_W-merge-cp^{**}\ T\ U \wedge conflict\ U\ V)$   
 $\vee (\exists\ S\ T. cdcl_W-merge-stgy^{**}\ R\ S \wedge no-step\ cdcl_W-merge-cp\ S \wedge decide\ S\ T$   
 $\wedge cdcl_W-merge-cp^{**}\ T\ V$   
 $\wedge conflicting\ V = C-True)$   
 $\vee (cdcl_W-merge-cp^{**}\ R\ V \wedge conflicting\ V = C-True)$   
 $\vee (\exists\ U. cdcl_W-merge-cp^{**}\ R\ U \wedge conflict\ U\ V)$   
**using** *assms(1,2)*

**proof** *induction*  
**case** *base*  
**then show** *?case* **by** *simp*

**next**  
**case** ( $step\ V\ W$ ) **note**  $st = this(1)$  **and**  $s' = this(2)$  **and**  $IH = this(3)[OF\ this(4)]$  **and**  
 $n-s-R = this(4)$   
**from**  $s'$   
**show** *?case*  
**proof** *cases*  
**case** *conflict'*  
**consider**  
 $(s')\ cdcl_W-merge-stgy^{**}\ R\ V$   
 $| (dec-conf)\ S\ T\ U$  **where**  $cdcl_W-merge-stgy^{**}\ R\ S$  **and**  $no-step\ cdcl_W-merge-cp\ S$  **and**  
 $decide\ S\ T$  **and**  $cdcl_W-merge-cp^{**}\ T\ U$  **and**  $conflict\ U\ V$   
 $| (dec)\ S\ T$  **where**  $cdcl_W-merge-stgy^{**}\ R\ S$  **and**  $no-step\ cdcl_W-merge-cp\ S$  **and**  $decide\ S\ T$   
**and**  $cdcl_W-merge-cp^{**}\ T\ V$  **and**  $conflicting\ V = C-True$   
 $| (cp)\ cdcl_W-merge-cp^{**}\ R\ V$   
 $| (cp-conf)\ U$  **where**  $cdcl_W-merge-cp^{**}\ R\ U$  **and**  $conflict\ U\ V$   
**using** *IH* **by** *meson*  
**then show** *?thesis*  
**proof** *cases*

```

next
  case  $s'$ 
  then have  $R = V$ 
    by (metis full1-def inv local.conflict' tranclp-unfold-begin
      rtranclp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
  consider
    (V-W)  $V = W$ 
    | (propa) propagate++  $V W$  and conflicting  $W = C\text{-True}$ 
    | (propa-conf)  $V'$  where propagate**  $V V'$  and conflict  $V' W$ 
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of  $V W$ ] conflict'
  unfolding full-unfold full1-def by blast
  then show ?thesis
  proof cases
    case V-W
    then show ?thesis using  $\langle R = V \rangle$  n-s-R by simp
  next
    case propa
    then show ?thesis using  $\langle R = V \rangle$  by auto
  next
    case propa-conf
    moreover
      then have cdclW-merge-cp**  $V V'$ 
      by (metis Nitpick.rtranclp-unfold cdclW-merge-cp.propagate' r-into-rtranclp)
    ultimately show ?thesis using  $s' \langle R = V \rangle$  by blast
  qed
next
  case dec-conf note - = this(5)
  then have False using conflict' unfolding full1-def by (auto dest!: tranclpD)
  then show ?thesis by fast
next
  case dec note  $T-V = \text{this}(4)$ 
  consider
    (propa) propagate++  $V W$  and conflicting  $W = C\text{-True}$ 
    | (propa-conf)  $V'$  where propagate**  $V V'$  and conflict  $V' W$ 
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of  $V W$ ] conflict'
  unfolding full1-def by blast
  then show ?thesis
  proof cases
    case propa
    then show ?thesis
    by (meson T-V cdclW-merge-cp.propagate' dec rtranclp.rtrancl-into-rtrancl)
  next
    case propa-conf
    then have cdclW-merge-cp**  $T V'$ 
    using T-V by (metis rtranclp-unfold cdclW-merge-cp.propagate' rtranclp.simps)
    then show ?thesis using dec propa-conf(2) by metis
  qed
next
  case cp
  consider
    (propa) propagate++  $V W$  and conflicting  $W = C\text{-True}$ 
    | (propa-conf)  $V'$  where propagate**  $V V'$  and conflict  $V' W$ 
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of  $V W$ ] conflict'
  unfolding full1-def by blast
  then show ?thesis

```

```

proof cases
  case propa
  then show ?thesis by (meson cdclW-merge-cp.propagate' cp rtranclp.rtrancl-into-rtrancl)
next
  case propa-conf
  then show ?thesis
    using propa-conf(2) by (metis rtranclp-unfold cdclW-merge-cp.propagate'
      cp rtranclp.rtrancl-into-rtrancl)
  qed
next
  case cp-conf
  then show ?thesis using conflict' unfolding full1-def by (fastforce dest!: tranclpD)
  qed
next
  case (decide' V')
  then have conf-V: conflicting V = C-True
    by auto
  consider
    (s') cdclW-merge-stgy** R V
  | (dec-conf) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (dec) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and decide S T
    and cdclW-merge-cp** T V and conflicting V = C-True
  | (cp) cdclW-merge-cp** R V
  | (cp-conf) U where cdclW-merge-cp** R U and conflict U V
  using IH by meson
  then show ?thesis
  proof cases
    case s'
    have conf-V': conflicting V' = C-True using decide'(1) by auto
    have full: full1 cdclW-cp V' W  $\vee$  (V' = W  $\wedge$  no-step cdclW-cp W)
      using decide'(3) unfolding full-unfold by blast
    consider
      (V'-W) V' = W
    | (propa) propagate++ V' W and conflicting W = C-True
    | (propa-conf) V'' where propagate** V' V'' and conflict V'' W
    using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V W] decide'
    by (metis (full1 cdclW-cp V' W  $\vee$  V' = W  $\wedge$  no-step cdclW-cp W) full1-def
      tranclp-cdclW-cp-propagate-with-conflict-or-not)
    then show ?thesis
  proof cases
    case V'-W
    then show ?thesis
      using conf-V' local.decide'(1,2) s' conf-V
      no-step-cdclW-cp-no-step-cdclW-merge-restart by auto
    next
    case propa
    then show ?thesis using local.decide'(1,2) s' by (metis cdclW-merge-cp.simps conf-V
      no-step-cdclW-cp-no-step-cdclW-merge-restart r-into-rtranclp)
    next
    case propa-conf
    then have cdclW-merge-cp** V' V''
      by (metis rtranclp-unfold cdclW-merge-cp.propagate' r-into-rtranclp)
    then show ?thesis
      using local.decide'(1,2) propa-conf(2) s' conf-V

```



```

      no-step-cdclW-cp-no-step-cdclW-merge-restart
    by metis
  qed
next
case (dec) note s' = this(1) and dec = this(2) and cp = this(3) and ns-cp-T = this(4)
have full cdclW-merge-cp T V
  unfolding full-def by (simp add: conf-V local.decide'(2)
    no-step-cdclW-cp-no-step-cdclW-merge-restart ns-cp-T)
moreover have no-step cdclW-merge-cp V
  by (simp add: conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart)
moreover have no-step cdclW-merge-cp S
  by (metis dec)
ultimately have cdclW-merge-stgy S V
  using cp by blast
then have cdclW-merge-stgy** R V using s' by auto
consider
  (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = C-True
  | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
  unfolding full-unfold full1-def by blast
then show ?thesis
proof cases
case V'-W
moreover have conflicting V' = C-True
  using decide'(1) by auto
ultimately show ?thesis
  using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩ by blast
next
case propa
moreover then have cdclW-merge-cp V' W
  by auto
ultimately show ?thesis
  using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩
  by (meson r-into-rtranclp)
next
case propa-confl
moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
qed
next
case cp
have no-step cdclW-merge-cp V
  using conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart by blast
then have full cdclW-merge-cp R V
  unfolding full-def using cp by fast
then have cdclW-merge-stgy** R V
  unfolding full-unfold by auto
have full1 cdclW-cp V' W ∨ (V' = W ∧ no-step cdclW-cp W)
  using decide'(3) unfolding full-unfold by blast

consider
  (V'-W) V' = W

```

```

| (propa) propagate++ V' W and conflicting W = C-True
| (propa-conf) V'' where propagate** V' V'' and conflict V'' W
using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
unfolding full-unfold full1-def by blast
then show ?thesis

proof cases
  case V'-W
  moreover have conflicting V' = C-True
  using decide'(1) by auto
  ultimately show ?thesis
  using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩ by blast
next
  case propa
  moreover then have cdclW-merge-cp V' W
  by auto
  ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
next
  case propa-conf
  moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
  ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
qed
next
  case (dec-conf)
  show ?thesis using conf-V dec-conf(5) by auto
next
  case cp-conf
  then show ?thesis using decide' by fastforce
qed
next
  case (bj' V')
  then have ¬no-step cdclW-bj V
  by (auto dest: tranclpD simp: full1-def)
  then consider
    (s') cdclW-merge-stgy** R V and conflicting V = C-True
  | (dec-conf) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (dec) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and decide S T
    and cdclW-merge-cp** T V and conflicting V = C-True
  | (cp) cdclW-merge-cp** R V and conflicting V = C-True
  | (cp-conf) U where cdclW-merge-cp** R U and conflict U V
  using IH by meson
then show ?thesis
proof cases
  case s' note - = this(2)
  then have False
  using bj'(1) unfolding full1-def by (force dest!: tranclpD simp: cdclW-bj.simps)
  then show ?thesis by fast
next
  case dec note - = this(5)
  then have False
  using bj'(1) unfolding full1-def by (force dest!: tranclpD simp: cdclW-bj.simps)

```

```

then show ?thesis by fast
next
case dec-confl
then have cdclW-merge-cp U V'
  using bj' cdclW-merge-cp.intros(1)[of U V V'] by (simp add: full-unfold)
then have cdclW-merge-cp** T V'
  using dec-confl(4) by simp
consider
  (V'-W) V' = W
| (propa) propagate++ V' W and conflicting W = C-True
| (propa-confl) V'' where propagate** V' V'' and conflict V'' W
using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] bj'(3)
unfolding full-unfold full1-def by blast
then show ?thesis
proof cases
case V'-W
then have no-step cdclW-cp V'
  using bj'(3) unfolding full-def by auto
then have no-step cdclW-merge-cp V'
  by (metis cdclW-cp.propagate' cdclW-merge-cp.cases tranclpD
    no-step-cdclW-cp-no-conflict-no-propagate(1) )
then have full1 cdclW-merge-cp T V'
  unfolding full1-def using ⟨cdclW-merge-cp U V'⟩ dec-confl(4) by auto
then have full cdclW-merge-cp T V'
  by (simp add: full-unfold)
then have cdclW-merge-stgy S V'
  using dec-confl(3) cdclW-merge-stgy.fw-s-decide ⟨no-step cdclW-merge-cp S⟩ by blast
then have cdclW-merge-stgy** R V'
  using ⟨cdclW-merge-stgy** R S⟩ by auto
show ?thesis
proof cases
assume conflicting W = C-True
then show ?thesis using ⟨cdclW-merge-stgy** R V'⟩ ⟨V' = W⟩ by auto
next
assume conflicting W ≠ C-True
then show ?thesis
  using ⟨cdclW-merge-stgy** R V'⟩ ⟨V' = W⟩ by (metis ⟨cdclW-merge-cp U V'⟩
    conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj dec-confl(5)
    r-into-rtranclp conflictE)
qed
next
case propa
moreover then have cdclW-merge-cp V' W
  by auto
ultimately show ?thesis using decide' by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3)
  rtranclp.rtrancl-into-rtrancl)
next
case propa-confl
moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
ultimately show ?thesis by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3) rtranclp-trans)
qed
next
case cp note - = this(2)
then show ?thesis using bj'(1) ⟨¬ no-step cdclW-bj V⟩

```

```

    conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj by auto
next
case cp-conf
then have cdclW-merge-cp U V' by (simp add: cdclW-merge-cp.conflict' full-unfold
    local.bj'(1))
consider
  (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = C-True
  | (propa-conf) V'' where propagate** V' V'' and conflict V'' W
using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] bj'
unfolding full-unfold full1-def by blast
then show ?thesis

proof cases
case V'-W
show ?thesis
proof cases
assume conflicting V' = C-True
then show ?thesis
  using V'-W ⟨cdclW-merge-cp U V'⟩ cp-conf(1) by force
next
assume confl: conflicting V' ≠ C-True
then have no-step cdclW-merge-stgy V'
  by (auto simp: cdclW-merge-stgy.simps full1-def full-def cdclW-merge-cp.simps
    dest!: tranclpD)
have no-step cdclW-merge-cp V'
  using confl by (auto simp: full1-def full-def cdclW-merge-cp.simps
    dest!: tranclpD)
moreover have cdclW-merge-cp U W
  using V'-W ⟨cdclW-merge-cp U V'⟩ by blast
ultimately have full1 cdclW-merge-cp R V'
  using cp-conf(1) V'-W unfolding full1-def by auto
then have cdclW-merge-stgy R V'
  by auto
moreover have no-step cdclW-merge-stgy V'
  using confl ⟨no-step cdclW-merge-cp V'⟩ by (auto simp: cdclW-merge-stgy.simps
    full1-def dest!: tranclpD)
ultimately have cdclW-merge-stgy** R V' by auto
show ?thesis by (metis V'-W ⟨cdclW-merge-cp U V'⟩ ⟨cdclW-merge-stgy** R V'⟩
  conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj cp-conf(1)
  rtranclp.rtrancl-into-rtrancl step.premis)
qed
next
case propa
moreover then have cdclW-merge-cp V' W
  by auto
ultimately show ?thesis using ⟨cdclW-merge-cp U V'⟩ cp-conf(1) by force
next
case propa-conf
moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
ultimately show ?thesis
  using ⟨cdclW-merge-cp U V'⟩ cp-conf(1) by (metis rtranclp.rtrancl-into-rtrancl
    rtranclp-trans)
qed

```

qed  
 qed  
 qed

**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*  
 $\bigwedge T. \text{ decide } S \ T \implies \text{no-step } \text{cdcl}_W\text{-merge-cp } S \implies \text{full } \text{cdcl}_W\text{-merge-cp } T \ U \implies P$   
**shows** *P*  
**using** *assms* **by** (*auto simp: cdcl<sub>W</sub>-merge-stgy.simps*)

**lemma** *decide-rtrancpl-cdcl<sub>W</sub>-s'-rtrancpl-cdcl<sub>W</sub>-s'*:

**assumes**  
*dec: decide S T and*  
*cdcl<sub>W</sub>-s<sup>'\*\*</sup> T U and*  
*n-s-S: no-step cdcl<sub>W</sub>-cp S and*  
*no-step cdcl<sub>W</sub>-cp U*  
**shows** *cdcl<sub>W</sub>-s<sup>'\*\*</sup> S U*  
**using** *assms(2,4)*

**proof** *induction*

**case** (*step U V*) **note** *st = this(1) and s' = this(2) and IH = this(3) and n-s = this(4)*

**consider**

(*TU*) *T = U*

| (*s'-st*) *T'* **where** *cdcl<sub>W</sub>-s' T T' and cdcl<sub>W</sub>-s<sup>'\*\*</sup> T' U*

**using** *st[unfolded rtrancpl-unfold] by (auto dest!: trancplD)*

**then show** *?case*

**proof** *cases*

**case** *TU*

**then show** *?thesis*

**proof** *–*

**have**  $\forall p \ s \ sa. (\neg p^{++} (s::'st) \ sa \vee (\exists sb. p^{**} \ s \ sb \wedge p \ sb \ sa))$

$\wedge ((\forall sb. \neg p^{**} \ s \ sb \vee \neg p \ sb \ sa) \vee p^{++} \ s \ sa)$

**by** (*metis trancpl-unfold-end*)

**then obtain** *ss :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st where*

*f2:  $\forall p \ s \ sa. (\neg p^{++} \ s \ sa \vee p^{**} \ s \ (ss \ p \ s \ sa) \wedge p \ (ss \ p \ s \ sa) \ sa)$*

$\wedge ((\forall sb. \neg p^{**} \ s \ sb \vee \neg p \ sb \ sa) \vee p^{++} \ s \ sa)$

**by** *moura*

**have** *f3: cdcl<sub>W</sub>-s' T V*

**using** *TU s' by blast*

**moreover**

{ **assume**  $\neg \text{cdcl}_W\text{-s}' \ S \ T$

**then have** *cdcl<sub>W</sub>-s' S V*

**using** *f3 by (metis (no-types) assms(1,3) cdcl<sub>W</sub>-s'.cases cdcl<sub>W</sub>-s'.decide' full-unfold)*

**then have** *cdcl<sub>W</sub>-s<sup>'++</sup> S V*

**by** *blast* }

**ultimately have** *cdcl<sub>W</sub>-s<sup>'++</sup> S V*

**using** *f2 by (metis (full-types) rtrancpl-unfold)*

**then show** *?thesis*

**by** *simp*

**qed**

**next**

**case** (*s'-st T'*) **note** *s'-T' = this(1) and st = this(2)*

**have** *cdcl<sub>W</sub>-s<sup>'\*\*</sup> S T'*

**using** *s'-T'*

```

proof cases
  case conflict'
  then have  $cdcl_W-s' S T'$ 
    using  $dec\ cdcl_W-s'.decide' n-s-S$  by ( $simp\ add: full-unfold$ )
  then show  $?thesis$ 
    using  $st$  by  $auto$ 
next
  case ( $decide' T''$ )
  then have  $cdcl_W-s' S T$ 
    using  $dec\ cdcl_W-s'.decide' n-s-S$  by ( $simp\ add: full-unfold$ )
  then show  $?thesis$  using  $decide' s'-T'$  by  $auto$ 
next
  case  $bj'$ 
  then have  $False$ 
    using  $dec\ unfolding\ full1-def$  by ( $fastforce\ dest!: tranclpD\ simp: cdcl_W-bj.simps$ )
  then show  $?thesis$  by  $fast$ 
qed
then show  $?thesis$  using  $s' st$  by  $auto$ 
qed
next
  case  $base$ 
  then have  $full\ cdcl_W-cp\ T\ T$ 
    by ( $simp\ add: full-unfold$ )
  then show  $?case$ 
    using  $cdcl_W-s'.simps\ dec\ n-s-S$  by  $auto$ 
qed

lemma  $rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-s'$ :
  assumes
     $cdcl_W-merge-stgy^{**} R\ V$  and
     $inv: cdcl_W-all-struct-inv\ R$ 
  shows  $cdcl_W-s'^{**} R\ V$ 
  using  $assms(1)$ 
proof induction
  case  $base$ 
  then show  $?case$  by  $simp$ 
next
  case ( $step\ S\ T$ ) note  $st = this(1)$  and  $fw = this(2)$  and  $IH = this(3)$ 
  have  $cdcl_W-all-struct-inv\ S$ 
    using  $inv\ rtranclp-cdcl_W-all-struct-inv-inv\ rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W\ st$  by  $blast$ 
from  $fw$  show  $?case$ 
  proof ( $cases\ rule: cdcl_W-merge-stgy-cases$ )
  case  $fw-s-cp$ 
  then show  $?thesis$ 
  proof –
    assume  $a1: full1\ cdcl_W-merge-cp\ S\ T$ 
    obtain  $ss :: ('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st$  where
       $f2: \bigwedge p\ s\ sa\ pa\ sb\ sc\ sd\ pb\ se\ sf. (\neg full1\ p\ (s::'st)\ sa \vee p^{++}\ s\ sa)$ 
       $\wedge (\neg pa\ (sb::'st)\ sc \vee \neg full1\ pa\ sd\ sb) \wedge (\neg pb^{++}\ se\ sf \vee pb\ sf\ (ss\ pb\ sf))$ 
       $\vee full1\ pb\ se\ sf)$ 
    by ( $metis\ (no-types)\ full1-def$ )
    then have  $f3: cdcl_W-merge-cp^{++}\ S\ T$ 
    using  $a1$  by  $auto$ 
    obtain  $ssa :: ('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st \Rightarrow 'st$  where
       $f4: \bigwedge p\ s\ sa. \neg p^{++}\ s\ sa \vee p\ s\ (ssa\ p\ s\ sa)$ 

```

```

    by (meson tranclp-unfold-begin)
  then have f5:  $\bigwedge s. \neg \text{full1 } \text{cdcl}_W\text{-merge-cp } s \ S$ 
    using f3 f2 by (metis (full-types))
  have  $\bigwedge s. \neg \text{full } \text{cdcl}_W\text{-merge-cp } s \ S$ 
    using f4 f3 by (meson full-def)
  then have  $S = R$ 
    using f5 by (metis (no-types)  $\text{cdcl}_W\text{-merge-stgy.simps } \text{rtranclp-unfold } st$ 
       $\text{tranclp-unfold-end}$ )
  then show ?thesis
    using f2 a1 by (metis (no-types)  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$ 
       $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-imp-full1-cdcl}_W\text{-s'-without-decode}$ 
       $\text{rtranclp-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W\text{-s' } \text{rtranclp-unfold}$ )
qed
next
case (fw-s-decide  $S'$ ) note dec = this(1) and n-S = this(2) and full = this(3)
moreover then have  $\text{conflicting } S' = C\text{-True}$ 
  by auto
ultimately have  $\text{full } \text{cdcl}_W\text{-s'-without-decide } S' \ T$ 
  by (meson  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$   $\text{cdcl}_W\text{-merge-restart-cdcl}_W$  fw-r-decide
     $\text{rtranclp-cdcl}_W\text{-all-struct-inv-inv}$ 
     $\text{conflicting-true-full-cdcl}_W\text{-merge-cp-iff-full-cdcl}_W\text{-s'-without-decode}$ )
then have a1:  $\text{cdcl}_W\text{-s}^{f**} S' \ T$ 
  unfolding full-def by (metis (full-types)  $\text{rtranclp-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W\text{-s'}$ )
have  $\text{cdcl}_W\text{-merge-stgy}^{f**} S \ T$ 
  using fw by blast
then have  $\text{cdcl}_W\text{-s}^{f**} S \ T$ 
  using decide-rtranclp-cdclW-s'-rtranclp-cdclW-s' a1 by (metis  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$  dec
    n-S no-step-cdclW-merge-cp-no-step-cdclW-cp  $\text{cdcl}_W\text{-all-struct-inv-def}$ 
     $\text{rtranclp-cdcl}_W\text{-merge-stgy'-no-step-cdcl}_W\text{-cp-or-eq}$ )
then show ?thesis using IH by auto
qed
qed

```

**lemma**  $\text{rtranclp-cdcl}_W\text{-merge-stgy-distinct-mset-clauses}$ :

```

  assumes invR:  $\text{cdcl}_W\text{-all-struct-inv } R$  and
    st:  $\text{cdcl}_W\text{-merge-stgy}^{f**} R \ S$  and
    dist:  $\text{distinct-mset } (\text{clauses } R)$  and
    R:  $\text{trail } R = []$ 
  shows  $\text{distinct-mset } (\text{clauses } S)$ 
  using  $\text{rtranclp-cdcl}_W\text{-stgy-distinct-mset-clauses}[OF \text{ invR } - \text{ dist } R]$ 
    invR st  $\text{rtranclp-mono}[of \text{ cdcl}_W\text{-s' } \text{cdcl}_W\text{-stgy}^{f**}]$   $\text{cdcl}_W\text{-s'-is-rtranclp-cdcl}_W\text{-stgy}$ 
  by (auto dest!:  $\text{cdcl}_W\text{-s'-is-rtranclp-cdcl}_W\text{-stgy } \text{rtranclp-cdcl}_W\text{-merge-stgy-rtranclp-cdcl}_W\text{-s'}$ )

```

**lemma**  $\text{no-step-cdcl}_W\text{-s'-no-step-cdcl}_W\text{-merge-stgy}$ :

```

  assumes
    inv:  $\text{cdcl}_W\text{-all-struct-inv } R$  and  $s'$ :  $\text{no-step } \text{cdcl}_W\text{-s' } R$ 
  shows  $\text{no-step } \text{cdcl}_W\text{-merge-stgy } R$ 

```

**proof** —

```

{ fix ss :: 'st
  obtain ssa :: 'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
    ff1:  $\bigwedge s \ sa. \neg \text{cdcl}_W\text{-merge-stgy } s \ sa \vee \text{full1 } \text{cdcl}_W\text{-merge-cp } s \ sa \vee \text{decide } s \ (ssa \ s \ sa)$ 
    using  $\text{cdcl}_W\text{-merge-stgy.cases}$  by moura
  obtain ssb :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
    ff2:  $\bigwedge p \ s \ sa. \neg p^{++} \ s \ sa \vee p \ s \ (ssb \ p \ s \ sa)$ 
    by (meson tranclp-unfold-begin)

```

```

obtain ssc :: 'st ⇒ 'st where
  ff3:  $\bigwedge s \text{ sa } sb. (\neg \text{cdcl}_W\text{-all-struct-inv } s \vee \neg \text{cdcl}_W\text{-cp } s \text{ sa} \vee \text{cdcl}_W\text{-s'} s (\text{ssc } s))$ 
     $\wedge (\neg \text{cdcl}_W\text{-all-struct-inv } s \vee \neg \text{cdcl}_W\text{-o } s \text{ sb} \vee \text{cdcl}_W\text{-s'} s (\text{ssc } s))$ 
  using n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o by moura
then have ff4:  $\bigwedge s. \neg \text{cdcl}_W\text{-o } R s$ 
  using s' inv by blast
have ff5:  $\bigwedge s. \neg \text{cdcl}_W\text{-cp}^{++} R s$ 
  using ff3 ff2 s' by (metis inv)
have  $\bigwedge s. \neg \text{cdcl}_W\text{-bj}^{++} R s$ 
  using ff4 ff2 by (metis bj)
then have  $\bigwedge s. \neg \text{cdcl}_W\text{-s'-without-decide } R s$ 
  using ff5 by (simp add: cdclW-s'-without-decide.simps full1-def)
then have  $\neg \text{cdcl}_W\text{-s'-without-decide}^{++} R ss$ 
  using ff2 by blast
then have  $\neg \text{cdcl}_W\text{-merge-stgy } R ss$ 
  using ff4 ff1 by (metis (full-types) decide full1-def inv
    conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode) }
then show ?thesis
  by fastforce
qed

lemma wf-cdclW-merge-cp:
  wf{(T, S).  $\text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S T$ }
  using wf-tranclp-cdclW-merge by (rule wf-subset) (auto simp: cdclW-merge-cp-tranclp-cdclW-merge)

lemma wf-cdclW-merge-stgy:
  wf{(T, S).  $\text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-stgy } S T$ }
  using wf-tranclp-cdclW-merge by (rule wf-subset)
  (auto simp add: cdclW-merge-stgy-tranclp-cdclW-merge)

lemma cdclW-merge-cp-obtain-normal-form:
  assumes inv:  $\text{cdcl}_W\text{-all-struct-inv } R$ 
  obtains S where full cdclW-merge-cp R S
proof –
  obtain S where full ( $\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S T$ ) R S
  using wf-exists-normal-form-full[OF wf-cdclW-merge-cp] by blast
then have
  st:  $(\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S T)^{**} R S$  and
  n-s: no-step ( $\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S T$ ) S
  unfolding full-def by blast+
have  $\text{cdcl}_W\text{-merge-cp}^{**} R S$ 
  using st by induction auto
moreover
  have  $\text{cdcl}_W\text{-all-struct-inv } S$ 
  using st inv
  apply (induction rule: rtranclp-induct)
  apply simp
  by (meson r-into-rtranclp rtranclp-cdclW-all-struct-inv-inv
    rtranclp-cdclW-merge-cp-rtranclp-cdclW)
then have no-step cdclW-merge-cp S
  using n-s by auto
ultimately show ?thesis
  using that unfolding full-def by blast
qed

```



**lemma** *no-step-cdcl<sub>W</sub>-merge-stgy-no-step-cdcl<sub>W</sub>-s'*:  
**assumes**  
*inv*: *cdcl<sub>W</sub>-all-struct-inv* *R* **and**  
*confl*: *conflicting* *R* = *C-True* **and**  
*n-s*: *no-step cdcl<sub>W</sub>-merge-stgy* *R*  
**shows** *no-step cdcl<sub>W</sub>-s'* *R*  
**proof** (*rule ccontr*)  
**assume**  $\neg$  *?thesis*  
**then obtain** *S* **where** *cdcl<sub>W</sub>-s'* *R S* **by** *auto*  
**then show** *False*  
**proof** *cases*  
**case** *conflict'*  
**then obtain** *S'* **where** *full1 cdcl<sub>W</sub>-merge-cp* *R S'*  
**by** (*metis* (*full-types*) *cdcl<sub>W</sub>-merge-cp-obtain-normal-form cdcl<sub>W</sub>-s'-without-decide.simps confl*  
*conflicting-true-no-step-cdcl<sub>W</sub>-merge-cp-no-step-s'-without-decide full-def full-unfold inv*  
*cdcl<sub>W</sub>-all-struct-inv-def*)  
**then show** *False* **using** *n-s* **by** *blast*  
**next**  
**case** (*decide' R'*)  
**then have** *cdcl<sub>W</sub>-all-struct-inv* *R'*  
**using** *inv cdcl<sub>W</sub>-all-struct-inv-inv cdcl<sub>W</sub>.other cdcl<sub>W</sub>-o.decide* **by** *meson*  
**then obtain** *R''* **where** *full cdcl<sub>W</sub>-merge-cp* *R' R''*  
**using** *cdcl<sub>W</sub>-merge-cp-obtain-normal-form* **by** *blast*  
**moreover have** *no-step cdcl<sub>W</sub>-merge-cp* *R*  
**by** (*simp add: confl local.decide'(2) no-step-cdcl<sub>W</sub>-cp-no-step-cdcl<sub>W</sub>-merge-restart*)  
**ultimately show** *False* **using** *n-s cdcl<sub>W</sub>-merge-stgy.intros local.decide'(1)* **by** *blast*  
**next**  
**case** (*bj' R'*)  
**then show** *False*  
**using** *confl no-step-cdcl<sub>W</sub>-cp-no-step-cdcl<sub>W</sub>-s'-without-decide inv*  
**unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *blast*  
**qed**  
**qed**

**lemma** *rtranclp-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-bj*:  
**assumes** *conflicting* *R* = *C-True* **and** *cdcl<sub>W</sub>-merge-cp\*\** *R S*  
**shows** *no-step cdcl<sub>W</sub>-bj* *S*  
**using** *assms conflicting-not-true-rtranclp-cdcl<sub>W</sub>-merge-cp-no-step-cdcl<sub>W</sub>-bj* **by** *blast*

**lemma** *rtranclp-cdcl<sub>W</sub>-merge-stgy-no-step-cdcl<sub>W</sub>-bj*:  
**assumes** *confl: conflicting* *R* = *C-True* **and** *cdcl<sub>W</sub>-merge-stgy\*\** *R S*  
**shows** *no-step cdcl<sub>W</sub>-bj* *S*  
**using** *assms(2)*  
**proof** *induction*  
**case** *base*  
**then show** *?case*  
**using** *confl* **by** (*auto simp: cdcl<sub>W</sub>-bj.simps*)[]  
**next**  
**case** (*step S T*) **note** *st = this(1)* **and** *fw = this(2)* **and** *IH = this(3)*  
**have** *confl-S: conflicting* *S* = *C-True*  
**using** *fw apply cases*  
**by** (*auto simp: full1-def cdcl<sub>W</sub>-merge-cp.simps dest!: tranclpD*)  
**from** *fw* **show** *?case*  
**proof** *cases*  
**case** *fw-s-cp*

```

    then show ?thesis
      using rtrancpl-cdclW-merge-cp-no-step-cdclW-bj confl-S
      by (simp add: full1-def trancpl-into-rtrancpl)
  next
    case (fw-s-decide S')
    moreover then have conflicting S' = C-True by auto
    ultimately show ?thesis
      using conflicting-not-true-rtrancpl-cdclW-merge-cp-no-step-cdclW-bj
      unfolding full-def by fast
  qed
qed

lemma full-cdclW-s'-full-cdclW-merge-restart:
  assumes
    conflicting R = C-True and
    inv: cdclW-all-struct-inv R
  shows full cdclW-s' R V  $\longleftrightarrow$  full cdclW-merge-stgy R V (is ?s'  $\longleftrightarrow$  ?fw)
proof
  assume ?s'
  then have cdclW-s'^** R V unfolding full-def by blast
  have cdclW-all-struct-inv V
    using ⟨cdclW-s'^** R V⟩ inv rtrancpl-cdclW-all-struct-inv-inv rtrancpl-cdclW-s'-rtrancpl-cdclW
    by blast
  then have n-s: no-step cdclW-merge-stgy V
    using no-step-cdclW-s'-no-step-cdclW-merge-stgy by (meson ⟨full cdclW-s' R V⟩ full-def)
  have n-s-bj: no-step cdclW-bj V
    by (metis ⟨cdclW-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ bj full-def
    n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  have n-s-cp: no-step cdclW-merge-cp V
  proof -
    { fix ss :: 'st
      obtain ssa :: 'st  $\Rightarrow$  'st where
        ff1:  $\forall s. \neg$  cdclW-all-struct-inv s  $\vee$  cdclW-s'-without-decide s (ssa s)
           $\vee$  no-step cdclW-merge-cp s
        using conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp by moura
      have ( $\forall p$  s sa. \neg full p (s::'st) sa  $\vee$  p** s sa  $\wedge$  no-step p sa) and
        ( $\forall p$  s sa. (\neg p** (s::'st) sa  $\vee$  ( $\exists s. p$  sa s))  $\vee$  full p sa)
        by (meson full-def)+
      then have  $\neg$  cdclW-merge-cp V ss
        using ff1 by (metis (no-types) ⟨cdclW-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ cdclW-s'.sims
          cdclW-s'-without-decide.cases) }
    then show ?thesis
      by blast
  }
  qed
consider
  (fw-no-confl) cdclW-merge-stgy** R V and conflicting V = C-True
| (fw-confl) cdclW-merge-stgy** R V and conflicting V  $\neq$  C-True and no-step cdclW-bj V
| (fw-dec-confl) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
  decide S T and cdclW-merge-cp** T U and conflict U V
| (fw-dec-no-confl) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
  decide S T and cdclW-merge-cp** T V and conflicting V = C-True
| (cp-no-confl) cdclW-merge-cp** R V and conflicting V = C-True
| (cp-confl) U where cdclW-merge-cp** R U and conflict U V
using rtrancpl-cdclW-s'-no-step-cdclW-s'-without-decide-decomp-into-cdclW-merge[OF
  ⟨cdclW-s'^** R V⟩ assms] by auto

```

```

then show ?fw
proof cases
  case fw-no-confl
  then show ?thesis using n-s unfolding full-def by blast
next
  case fw-confl
  then show ?thesis using n-s unfolding full-def by blast
next
  case fw-dec-confl
  have cdclW-merge-cp U V
    using n-s-bj by (metis cdclW-merge-cp.simps full-unfold fw-dec-confl(5))
  then have full1 cdclW-merge-cp T V
    unfolding full1-def by (metis fw-dec-confl(4) n-s-cp tranclp-unfold-end)
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case fw-dec-no-confl
  then have full cdclW-merge-cp T V
    using n-s-cp unfolding full-def by blast
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case cp-no-confl
  then have full cdclW-merge-cp R V
    by (simp add: full-def n-s-cp)
  then have R = V ∨ cdclW-merge-stgy++ R V
    by (metis (no-types) full-unfold fw-s-cp rtranclp-unfold tranclp-unfold-end)
  then show ?thesis
    by (simp add: full-def n-s rtranclp-unfold)
next
  case cp-confl
  have full cdclW-bj V V
    using n-s-bj unfolding full-def by blast
  then have full1 cdclW-merge-cp R V
    unfolding full1-def by (meson cdclW-merge-cp.conflict' cp-confl(1,2) n-s-cp
      rtranclp-into-tranclp1)
  then show ?thesis using n-s unfolding full-def by auto
qed
next
assume ?fw
then have cdclW** R V using rtranclp-mono[of cdclW-merge-stgy cdclW**]
  cdclW-merge-stgy-rtranclp-cdclW unfolding full-def by auto
then have inv': cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
have cdclW-s'** R V
  using ⟨?fw⟩ by (simp add: full-def inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s')
moreover have no-step cdclW-s' V
proof cases
  assume conflicting V = C-True
  then show ?thesis
    by (metis inv' ⟨full cdclW-merge-stgy R V⟩ full-def
      no-step-cdclW-merge-stgy-no-step-cdclW-s')
next
  assume confl-V: conflicting V ≠ C-True
  then have no-step cdclW-bj V
    using rtranclp-cdclW-merge-stgy-no-step-cdclW-bj by (meson ⟨full cdclW-merge-stgy R V⟩

```

```

    assms(1) full-def)
  then show ?thesis using confl-V by (fastforce simp: cdclW-s'.simps full1-def cdclW-cp.simps
    dest!: tranclpD)
qed
ultimately show ?s' unfolding full-def by blast
qed

```

**lemma** *full-cdcl<sub>W</sub>-stgy-full-cdcl<sub>W</sub>-merge:*

```

  assumes
    conflicting R = C-True and
    inv: cdclW-all-struct-inv R
  shows full cdclW-stgy R V  $\longleftrightarrow$  full cdclW-merge-stgy R V
  by (simp add: assms(1) full-cdclW-s'-full-cdclW-merge-restart full-cdclW-stgy-iff-full-cdclW-s'
    inv)

```

**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' = C-Clause {#}  $\wedge$  unsatisfiable (set-mset N))
     $\vee$  (conflicting S' = C-True  $\wedge$  trail S'  $\models_{asm}$  N  $\wedge$  satisfiable (set-mset N))

```

**proof** –

```

  have cdclW-all-struct-inv (init-state N)
    using no-d unfolding cdclW-all-struct-inv-def by auto
  moreover have conflicting (init-state N) = C-True
    by auto
  ultimately show ?thesis
    by (simp add: full full-cdclW-stgy-final-state-conclusive-from-init-state
      full-cdclW-stgy-full-cdclW-merge no-d)

```

**qed**

**end**

## 19.5 Adding Restarts

**locale** *cdcl<sub>W</sub>-ops-restart =*

```

  cdclW-ops trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
  add-init-clss
  add-learned-clss remove-clss update-backtrack-lvl update-conflicting init-state
  restart-state

```

**for**

```

  trail :: 'st  $\Rightarrow$  ('v::linorder, nat, 'v clause) marked-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 conflicting-clause and

```

```

  cons-trail :: ('v, nat, 'v clause) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-init-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  add-learned-clss remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause conflicting-clause  $\Rightarrow$  'st  $\Rightarrow$  'st and

```

```

  init-state :: 'v::linorder clauses  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st +

```

```

fixes f :: nat ⇒ nat
assumes f: unbounded f
begin

```

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

**inductive** *cdcl<sub>W</sub>-merge-with-restart* **where**

*restart-step*:

```

(cyclW-merge-stgy ~ (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) S T
⇒ card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > f n
⇒ restart T U ⇒ cyclW-merge-with-restart (S, n) (U, Suc n) |

```

*restart-full*: *full1 cycl<sub>W</sub>-merge-stgy S T ⇒ cycl<sub>W</sub>-merge-with-restart (S, n) (T, Suc n)*

**lemma** *cdcl<sub>W</sub>-merge-with-restart S T ⇒ cycl<sub>W</sub>-merge-restart\*\* (fst S) (fst T)*

**by** (*induction rule: cycl<sub>W</sub>-merge-with-restart.induct*)

```

(auto dest!: relpowp-imp-rtranclp cyclW-merge-stgy-tranclp-cyclW-merge tranclp-into-rtranclp
  rtranclp-cyclW-merge-stgy-rtranclp-cyclW-merge rtranclp-cyclW-merge-tranclp-cyclW-merge-restart
  fw-r-rf cyclW-rf.restart
  simp: full1-def)

```

**lemma** *cdcl<sub>W</sub>-merge-with-restart-rtranclp-cycl<sub>W</sub>:*

*cdcl<sub>W</sub>-merge-with-restart S T ⇒ cycl<sub>W</sub>\*\* (fst S) (fst T)*

**by** (*induction rule: cycl<sub>W</sub>-merge-with-restart.induct*)

```

(auto dest!: relpowp-imp-rtranclp rtranclp-cyclW-merge-stgy-rtranclp-cyclW cyclW.rf
  cyclW-rf.restart tranclp-into-rtranclp simp: full1-def)

```

**lemma** *cdcl<sub>W</sub>-merge-with-restart-increasing-number:*

*cdcl<sub>W</sub>-merge-with-restart S T ⇒ snd T = 1 + snd S*

**by** (*induction rule: cycl<sub>W</sub>-merge-with-restart.induct*) *auto*

**lemma** *full1 cycl<sub>W</sub>-merge-stgy S T ⇒ cycl<sub>W</sub>-merge-with-restart (S, n) (T, Suc n)*

**using** *restart-full* **by** *blast*

**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) ⊆ build-all-simple-clss (atms-of-mu (init-clss S))*

**proof**

**fix** *C*

**assume** *C: C ∈ set-mset (learned-clss S)*

**have** *distinct-mset C*

**using** *C inv unfolding cdcl<sub>W</sub>-all-struct-inv-def distinct-cycl<sub>W</sub>-state-def distinct-mset-set-def*

**by** *auto*

**moreover** **have** *¬tautology C*

**using** *C inv unfolding cdcl<sub>W</sub>-all-struct-inv-def cdcl<sub>W</sub>-learned-clause-def* **by** *auto*

**moreover**

**have** *atms-of C ⊆ atms-of-mu (learned-clss S)*

**using** *C* **by** *auto*

**then** **have** *atms-of C ⊆ atms-of-mu (init-clss S)*

**using** *inv unfolding cdcl<sub>W</sub>-all-struct-inv-def no-strange-atm-def* **by** *force*

**moreover** **have** *finite (atms-of-mu (init-clss S))*

**using** *inv unfolding cdcl<sub>W</sub>-all-struct-inv-def* **by** *auto*

**ultimately** **show** *C ∈ build-all-simple-clss (atms-of-mu (init-clss S))*

**using** *distinct-mset-not-tautology-implies-in-build-all-simple-clss build-all-simple-clss-mono*

**by** *blast*

qed

**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)*

**using** *cdcl<sub>W</sub>-merge-with-restart-rtrancpl-cdcl<sub>W</sub> rtrancpl-cdcl<sub>W</sub>-init-clss* **by** *blast*

**lemma**

*wf {(T, S). cdcl<sub>W</sub>-all-struct-inv (fst S)  $\wedge$  cdcl<sub>W</sub>-merge-with-restart S T}*

**proof** (*rule ccontr*)

**assume**  $\neg$  *?thesis*

**then obtain g where**

*g:  $\bigwedge i$ . cdcl<sub>W</sub>-merge-with-restart (g i) (g (Suc i))* **and**

*inv:  $\bigwedge i$ . cdcl<sub>W</sub>-all-struct-inv (fst (g i))*

**unfolding** *wf-iff-no-infinite-down-chain* **by** *fast*

**{ fix i**

**have** *init-clss (fst (g i)) = init-clss (fst (g 0))*

**apply** (*induction i*)

**apply** *simp*

**using** *g inv unfolding cdcl<sub>W</sub>-all-struct-inv-def* **by** (*metis cdcl<sub>W</sub>-merge-with-restart-init-clss*)

**} note** *init-g = this*

**let** *?S = g 0*

**have** *finite (atms-of-mu (init-clss (fst ?S)))*

**using** *inv unfolding cdcl<sub>W</sub>-all-struct-inv-def* **by** *auto*

**have** *snd-g:  $\bigwedge i$ . snd (g i) = i + snd (g 0)*

**apply** (*induct-tac i*)

**apply** *simp*

**by** (*metis Suc-eq-plus1-left add-Suc cdcl<sub>W</sub>-merge-with-restart-increasing-number g*)

**then have** *snd-g-0:  $\bigwedge i$ . i > 0  $\implies$  snd (g i) = i + snd (g 0)*

**by** *blast*

**have** *unbounded-f-g: unbounded ( $\lambda i$ . f (snd (g i)))*

**using** *f unfolding bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*

*not-bounded-nat-exists-larger not-le ordered-cancel-comm-monoid-diff-class.le-iff-add*)

**obtain k where**

*f-g-k: f (snd (g k)) > card (build-all-simple-clss (atms-of-mu (init-clss (fst ?S))))* **and**

*k > card (build-all-simple-clss (atms-of-mu (init-clss (fst ?S))))*

**using** *not-bounded-nat-exists-larger[OF unbounded-f-g]* **by** *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

**{ fix i**

**assume** *no-step cdcl<sub>W</sub>-merge-stgy (fst (g i))*

**with** *g[of i]*

**have** *False*

**proof** (*induction rule: cdcl<sub>W</sub>-merge-with-restart.induct*)

**case** (*restart-step T S n*) **note** *H = this(1)* **and** *c = this(2)* **and** *n-s = this(4)*

**obtain S' where** *cdcl<sub>W</sub>-merge-stgy S S'*

**using** *H c* **by** (*metis gr-implies-not0 relpowp-E2*)

**then show** *False* **using** *n-s* **by** *auto*

**next**

**case** (*restart-full S T*)

**then show** *False* **unfolding** *full1-def* **by** (*auto dest: trancplD*)

**qed**

**} note** *H = this*

**obtain m T where**

$m: m = \text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss} (\text{fst } (g \ k))))$  **and**  
 $m > f (\text{snd } (g \ k))$  **and**  
 $\text{restart } T (\text{fst } (g \ (k+1)))$  **and**  
 $\text{cdcl}_W\text{-merge-stgy}: (\text{cdcl}_W\text{-merge-stgy} \rightsquigarrow m) (\text{fst } (g \ k)) \ T$   
**using**  $g[\text{of } k] \ H[\text{of } \text{Suc } k]$  **by**  $(\text{force simp: cdcl}_W\text{-merge-with-restart.simps full1-def})$   
**have**  $\text{cdcl}_W\text{-merge-stgy}^{**} (\text{fst } (g \ k)) \ T$   
**using**  $\text{cdcl}_W\text{-merge-stgy relpowp-imp-rtrancpl}$  **by**  $\text{metis}$   
**then have**  $\text{cdcl}_W\text{-all-struct-inv } T$   
**using**  $\text{inv}[\text{of } k] \ \text{rtrancpl-cdcl}_W\text{-all-struct-inv-inv rtrancpl-cdcl}_W\text{-merge-stgy-rtrancpl-cdcl}_W$   
**by**  $\text{blast}$   
**moreover have**  $\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss} (\text{fst } (g \ k))))$   
 $> \text{card} (\text{build-all-simple-clss} (\text{atms-of-mu} (\text{init-clss} (\text{fst } ?S))))$   
**unfolding**  $m[\text{symmetric}]$  **using**  $\langle m > f (\text{snd } (g \ k)) \rangle \ f\text{-}g\text{-}k$  **by**  $\text{linarith}$   
**then have**  $\text{card} (\text{set-mset} (\text{learned-clss } T))$   
 $> \text{card} (\text{build-all-simple-clss} (\text{atms-of-mu} (\text{init-clss} (\text{fst } ?S))))$   
**by**  $\text{linarith}$   
**moreover**  
**have**  $\text{init-clss} (\text{fst } (g \ k)) = \text{init-clss } T$   
**using**  $\langle \text{cdcl}_W\text{-merge-stgy}^{**} (\text{fst } (g \ k)) \ T \rangle \ \text{rtrancpl-cdcl}_W\text{-merge-stgy-rtrancpl-cdcl}_W$   
 $\text{rtrancpl-cdcl}_W\text{-init-clss inv}$  **unfolding**  $\text{cdcl}_W\text{-all-struct-inv-def}$  **by**  $\text{blast}$   
**then have**  $\text{init-clss} (\text{fst } ?S) = \text{init-clss } T$   
**using**  $\text{init-g}[\text{of } k]$  **by**  $\text{auto}$   
**ultimately show**  $\text{False}$   
**using**  $\text{cdcl}_W\text{-all-struct-inv-learned-clss-bound}$  **by**  $(\text{metis Suc-leI card-mono not-less-eq-eq}$   
 $\text{build-all-simple-clss-finite})$   
**qed**

**lemma**  $\text{cdcl}_W\text{-merge-with-restart-distinct-mset-clauses}$ :

**assumes**  $\text{invR}: \text{cdcl}_W\text{-all-struct-inv} (\text{fst } R)$  **and**  
 $\text{st}: \text{cdcl}_W\text{-merge-with-restart } R \ S$  **and**  
 $\text{dist}: \text{distinct-mset} (\text{clauses} (\text{fst } R))$  **and**  
 $R: \text{trail} (\text{fst } R) = []$   
**shows**  $\text{distinct-mset} (\text{clauses} (\text{fst } S))$   
**using**  $\text{assms}(2,1,3,4)$   
**proof**  $(\text{induction})$   
**case**  $(\text{restart-full } S \ T)$   
**then show**  $?case$  **using**  $\text{rtrancpl-cdcl}_W\text{-merge-stgy-distinct-mset-clauses}[\text{of } S \ T]$  **unfolding**  $\text{full1-def}$   
**by**  $(\text{auto dest: rtrancpl-into-rtrancpl})$   
**next**  
**case**  $(\text{restart-step } T \ S \ n \ U)$   
**then have**  $\text{distinct-mset} (\text{clauses } T)$   
**using**  $\text{rtrancpl-cdcl}_W\text{-merge-stgy-distinct-mset-clauses}[\text{of } S \ T]$  **unfolding**  $\text{full1-def}$   
**by**  $(\text{auto dest: relpowp-imp-rtrancpl})$   
**then show**  $?case$  **using**  $\langle \text{restart } T \ U \rangle$  **by**  $(\text{metis clauses-restart distinct-mset-union fstI}$   
 $\text{mset-le-exists-conv restart.cases state-eq-clauses})$   
**qed**

**inductive**  $\text{cdcl}_W\text{-with-restart}$  **where**

$\text{restart-step}$ :

$(\text{cdcl}_W\text{-stgy} \rightsquigarrow (\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss } S)))) \ S \ T \implies$   
 $\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss } S)) > f \ n \implies$   
 $\text{restart } T \ U \implies$   
 $\text{cdcl}_W\text{-with-restart} (S, n) (U, \text{Suc } n) \mid$   
 $\text{restart-full: full1 cdcl}_W\text{-stgy } S \ T \implies \text{cdcl}_W\text{-with-restart} (S, n) (T, \text{Suc } n)$

```

lemma cdclW-with-restart-rtrancp-cdclW:
  cdclW-with-restart S T  $\implies$  cdclW** (fst S) (fst T)
apply (induction rule: cdclW-with-restart.induct)
by (auto dest!: relopw-imp-rtrancp trancp-into-rtrancp fw-r-rf
  cdclW-rf.restart rtrancp-cdclW-stgy-rtrancp-cdclW cdclW-merge-restart-cdclW
  simp: full1-def)

lemma cdclW-with-restart-increasing-number:
  cdclW-with-restart S T  $\implies$  snd T = 1 + snd S
by (induction rule: cdclW-with-restart.induct) auto

lemma full1 cdclW-stgy S T  $\implies$  cdclW-with-restart (S, n) (T, Suc n)
  using restart-full by blast

lemma cdclW-with-restart-init-clss:
  cdclW-with-restart S T  $\implies$  cdclW-M-level-inv (fst S)  $\implies$  init-clss (fst S) = init-clss (fst T)
  using cdclW-with-restart-rtrancp-cdclW rtrancp-cdclW-init-clss by blast

lemma
  wf {(T, S). cdclW-all-struct-inv (fst S)  $\wedge$  cdclW-with-restart S T}
proof (rule ccontr)
assume  $\neg$  ?thesis
  then obtain g where
    g:  $\bigwedge i.$  cdclW-with-restart (g i) (g (Suc i)) and
    inv:  $\bigwedge i.$  cdclW-all-struct-inv (fst (g i))
  unfolding wf-iff-no-infinite-down-chain by fast
  { fix i
    have init-clss (fst (g i)) = init-clss (fst (g 0))
      apply (induction i)
      apply simp
      using g inv unfolding cdclW-all-struct-inv-def by (metis cdclW-with-restart-init-clss)
    } note init-g = this
  let ?S = g 0
  have finite (atms-of-mu (init-clss (fst ?S)))
    using inv unfolding cdclW-all-struct-inv-def by auto
  have snd-g:  $\bigwedge i.$  snd (g i) = i + snd (g 0)
    apply (induct-tac i)
    apply simp
    by (metis Suc-eq-plus1-left add-Suc cdclW-with-restart-increasing-number g)
  then have snd-g-0:  $\bigwedge i.$  i > 0  $\implies$  snd (g i) = i + snd (g 0)
    by blast
  have unbounded-f-g: unbounded ( $\lambda i.$  f (snd (g i)))
    using f unfolding bounded-def by (metis add.commute f less-or-eq-imp-le snd-g
      not-bounded-nat-exists-larger not-le ordered-cancel-comm-monoid-diff-class.le-iff-add)

  obtain k where
    f-g-k: f (snd (g k)) > card (build-all-simple-clss (atms-of-mu (init-clss (fst ?S)))) and
    k > card (build-all-simple-clss (atms-of-mu (init-clss (fst ?S))))
    using not-bounded-nat-exists-larger[OF unbounded-f-g] by blast

```

The following does not hold anymore with the non-strict version of cardinality in the definition.

```

{ fix i
  assume no-step cdclW-stgy (fst (g i))
  with g[of i]
  have False

```



```

proof (induction rule: cdclW-with-restart.induct)
  case (restart-step T S n) note H = this(1) and c = this(2) and n-s = this(4)
  obtain S' where cdclW-stgy S S'
    using H c by (metis gr-implies-not0 relpowp-E2)
  then show False using n-s by auto
next
  case (restart-full S T)
  then show False unfolding full1-def by (auto dest: tranclpD)
qed
} note H = this
obtain m T where
  m: m = card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k)))) and
  m > f (snd (g k)) and
  restart T (fst (g (k+1))) and
  cdclW-merge-stgy: (cdclW-stgy  $\sim$  m) (fst (g k)) T
  using g[of k] H[of Suc k] by (force simp: cdclW-with-restart.simps full1-def)
have cdclW-stgy** (fst (g k)) T
  using cdclW-merge-stgy relpowp-imp-rtranclp by metis
then have cdclW-all-struct-inv T
  using inv[of k] rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW by blast
moreover have card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k))))
  > card (build-all-simple-clss (atms-of-mu (init-clss (fst ?S))))
  unfolding m[symmetric] using m > f (snd (g k)) f-g-k by linarith
then have card (set-mset (learned-clss T))
  > card (build-all-simple-clss (atms-of-mu (init-clss (fst ?S))))
  by linarith
moreover
  have init-clss (fst (g k)) = init-clss T
  using cdclW-stgy** (fst (g k)) T rtranclp-cdclW-stgy-rtranclp-cdclW rtranclp-cdclW-init-clss
  inv unfolding cdclW-all-struct-inv-def
  by blast
  then have init-clss (fst ?S) = init-clss T
  using init-g[of k] by auto
ultimately show False
  using cdclW-all-struct-inv-learned-clss-bound by (metis Suc-leI card-mono not-less-eq-eq
    build-all-simple-clss-finite)
qed

lemma cdclW-with-restart-distinct-mset-clauses:
  assumes invR: cdclW-all-struct-inv (fst R) and
  st: cdclW-with-restart R S and
  dist: distinct-mset (clauses (fst R)) and
  R: trail (fst R) = []
  shows distinct-mset (clauses (fst S))
  using assms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtranclp-cdclW-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: tranclp-into-rtranclp)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T) using rtranclp-cdclW-stgy-distinct-mset-clauses[of S T]
    unfolding full1-def by (auto dest: relpowp-imp-rtranclp)
  then show ?case using restart T U by (metis clauses-restart distinct-mset-union fstI
    mset-le-exists-conv restart.cases state-eq-clauses)

```

```

qed
end

locale luby-sequence =
  fixes ur :: nat
  assumes ur > 0
begin

lemma exists-luby-decomp:
  fixes i :: nat
  shows  $\exists k::nat. (2^{k-1} \leq i \wedge i < 2^k - 1) \vee i = 2^k - 1$ 
proof (induction i)
  case 0
  then show ?case
    by (rule exI[of - 0], simp)
next
  case (Suc n)
  then obtain k where  $2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1$ 
    by blast
  then consider
    (st-interv)  $2^{k-1} \leq n$  and  $n \leq 2^k - 2$ 
  | (end-interv)  $2^{k-1} \leq n$  and  $n = 2^k - 2$ 
  | (pow2)  $n = 2^k - 1$ 
    by linarith
  then show ?case
    proof cases
    case st-interv
    then show ?thesis apply - apply (rule exI[of - k])
      by (metis (no-types, lifting) One-nat-def Suc-diff-Suc Suc-lessI
         $(2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1)$  diff-self-eq-0
        dual-order.trans le-SucI le-imp-less-Suc numeral-2-eq-2 one-le-numeral
        one-le-power zero-less-numeral zero-less-power)
    next
    case end-interv
    then show ?thesis apply - apply (rule exI[of - k]) by auto
    next
    case pow2
    then show ?thesis apply - apply (rule exI[of - k+1]) by auto
  qed
qed

```

Luby sequences are defined by:

- $2^k - 1$ , if  $i = (2::'a)^k - (1::'a)$
- $\text{luby-sequence-core } (i - 2^{k-1} + 1)$ , 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  $\exists k. i = 2^k - 1$ 
     then  $2^{((\text{SOME } k. i = 2^k - 1) - 1)}$ 
     else luby-sequence-core (i -  $2^{((\text{SOME } k. 2^{k-1} \leq i \wedge i < 2^k - 1) - 1) + 1}$ ))
  by auto
termination

```

```

proof (relation less-than, goal-cases)
  case 1
  then show ?case by auto
next
  case (2 i)
  let ?k = (SOME k.  $2 \wedge (k - 1) \leq i \wedge i < 2 \wedge k - 1$ )
  have  $2 \wedge (?k - 1) \leq i \wedge i < 2 \wedge ?k - 1$ 
  apply (rule someI-ex)
  using 2 exists-luby-decomp by blast
  then show ?case

proof –
  have  $\forall n \text{ na. } \neg (1::\text{nat}) \leq n \vee 1 \leq n \wedge \text{na}$ 
  by (meson one-le-power)
  then have f1:  $(1::\text{nat}) \leq 2 \wedge (?k - 1)$ 
  using one-le-numeral by blast
  have f2:  $i - 2 \wedge (?k - 1) + 2 \wedge (?k - 1) = i$ 
  using  $2 \wedge (?k - 1) \leq i \wedge i < 2 \wedge ?k - 1$  le-add-diff-inverse2 by blast
  have f3:  $2 \wedge ?k - 1 \neq \text{Suc } 0$ 
  using f1  $2 \wedge (?k - 1) \leq i \wedge i < 2 \wedge ?k - 1$  by linarith
  have  $2 \wedge ?k - (1::\text{nat}) \neq 0$ 
  using  $2 \wedge (?k - 1) \leq i \wedge i < 2 \wedge ?k - 1$  gr-implies-not0 by blast
  then have f4:  $2 \wedge ?k \neq (1::\text{nat})$ 
  by linarith
  have f5:  $\forall n \text{ na. if } \text{na} = 0 \text{ then } (n::\text{nat}) \wedge \text{na} = 1 \text{ else } n \wedge \text{na} = n * n \wedge (\text{na} - 1)$ 
  by (simp add: power-eq-if)
  then have ?k  $\neq 0$ 
  using f4 by meson
  then have  $2 \wedge (?k - 1) \neq \text{Suc } 0$ 
  using f5 f3 by presburger
  then have  $\text{Suc } 0 < 2 \wedge (?k - 1)$ 
  using f1 by linarith
  then show ?thesis
  using f2 less-than-iff by presburger
qed
qed

declare luby-sequence-core.simps[simp del]

lemma two-pover-n-eq-two-power-n'-eq:
  assumes H:  $(2::\text{nat}) \wedge (k::\text{nat}) - 1 = 2 \wedge k' - 1$ 
  shows  $k' = k$ 
proof –
  have  $(2::\text{nat}) \wedge (k::\text{nat}) = 2 \wedge k'$ 
  using H by (metis One-nat-def Suc-pred zero-less-numeral zero-less-power)
  then show ?thesis by simp
qed

lemma luby-sequence-core-two-power-minus-one:
  luby-sequence-core  $(2 \wedge k - 1) = 2 \wedge (k - 1)$  (is ?L = ?K)
proof –
  have decomp:  $\exists ka. 2 \wedge k - 1 = 2 \wedge ka - 1$ 
  by auto
  have ?L =  $2 \wedge ((\text{SOME } k'. (2::\text{nat}) \wedge k - 1 = 2 \wedge k' - 1) - 1)$ 
  apply (subst luby-sequence-core.simps, subst decomp)

```

```

  by simp
moreover have (SOME k'. (2::nat) ^ k - 1 = 2 ^ k' - 1) = k
  apply (rule some-equality)
  apply simp
  using two-pover-n-eq-two-power-n'-eq by blast
ultimately show ?thesis by presburger
qed

```

**lemma** *different-luby-decomposition-false:*

```

assumes
  H: 2 ^ (k - Suc 0) ≤ i and
  k': i < 2 ^ k' - Suc 0 and
  k-k': k > k'
shows False
proof -
  have 2 ^ k' - Suc 0 < 2 ^ (k - Suc 0)
    using k-k' less-eq-Suc-le by auto
  then show ?thesis
    using H k' by linarith
qed

```

**lemma** *luby-sequence-core-not-two-power-minus-one:*

```

assumes
  k-i: 2 ^ (k - 1) ≤ i and
  i-k: i < 2 ^ k - 1
shows luby-sequence-core i = luby-sequence-core (i - 2 ^ (k - 1) + 1)
proof -
  have H: ¬ (∃ ka. i = 2 ^ ka - 1)
  proof (rule ccontr)
    assume ¬ ?thesis
    then obtain k':nat where k': i = 2 ^ k' - 1 by blast
    have (2::nat) ^ k' - 1 < 2 ^ k - 1
      using i-k unfolding k' .
    then have (2::nat) ^ k' < 2 ^ k
      by linarith
    then have k' < k
      by simp
    have 2 ^ (k - 1) ≤ 2 ^ k' - (1::nat)
      using k-i unfolding k' .
    then have (2::nat) ^ (k-1) < 2 ^ k'
      by (metis Suc-diff-1 not-le not-less-eq zero-less-numeral zero-less-power)
    then have k-1 < k'
      by simp

    show False using (k' < k) (k-1 < k') by linarith
  qed
  have ∧ k k'. 2 ^ (k - Suc 0) ≤ i ⇒ i < 2 ^ k - Suc 0 ⇒ 2 ^ (k' - Suc 0) ≤ i ⇒
    i < 2 ^ k' - Suc 0 ⇒ k = k'
    by (meson different-luby-decomposition-false linorder-neqE-nat)
  then have k: (SOME k. 2 ^ (k - Suc 0) ≤ i ∧ i < 2 ^ k - Suc 0) = k
    using k-i i-k by auto
  show ?thesis
    apply (subst luby-sequence-core.simps[of i], subst H)
    by (simp add: k)
qed

```

**lemma** *unbounded-luby-sequence-core: unbounded luby-sequence-core*  
**unfolding** *bounded-def*  
**proof**  
**assume**  $\exists b. \forall n. \text{luby-sequence-core } n \leq b$   
**then obtain**  $b$  **where**  $b: \bigwedge n. \text{luby-sequence-core } n \leq b$   
**by** *metis*  
**have**  $\text{luby-sequence-core } (2^{b+1} - 1) = 2^b$   
**using** *luby-sequence-core-two-power-minus-one[of b+1]* **by** *simp*  
**moreover have**  $(2::\text{nat})^b > b$   
**by** *(induction b) auto*  
**ultimately show** *False* **using**  $b[\text{of } 2^{b+1} - 1]$  **by** *linarith*  
**qed**

**abbreviation** *luby-sequence :: nat  $\Rightarrow$  nat* **where**  
*luby-sequence*  $n \equiv ur * \text{luby-sequence-core } n$

**lemma** *bounded-luby-sequence: unbounded luby-sequence*  
**using** *bounded-const-product[of ur] luby-sequence-axioms*  
*luby-sequence-def unbounded-luby-sequence-core* **by** *blast*

**lemma** *luby-sequence-core-0: luby-sequence-core 0 = 1*  
**proof** –  
**have**  $0: (0::\text{nat}) = 2^0 - 1$   
**by** *auto*  
**show** *?thesis*  
**by** *(subst 0, subst luby-sequence-core-two-power-minus-one) simp*  
**qed**

**lemma** *luby-sequence-core n  $\geq$  1*  
**proof** *(induction n rule: nat-less-induct-case)*  
**case**  $0$   
**then show** *?case* **by** *(simp add: luby-sequence-core-0)*  
**next**  
**case** *(Suc n)* **note** *IH = this*

**consider**  
*(interv) k* **where**  $2^k - 1 \leq \text{Suc } n$  **and**  $\text{Suc } n < 2^{k+1} - 1$   
 $|$  *(pow2) k* **where**  $\text{Suc } n = 2^k - \text{Suc } 0$   
**using** *exists-luby-decomp[of Suc n]* **by** *auto*

**then show** *?case*  
**proof** *cases*  
**case** *pow2*  
**show** *?thesis*  
**using** *luby-sequence-core-two-power-minus-one pow2* **by** *auto*  
**next**  
**case** *interv*  
**have**  $n: \text{Suc } n - 2^k + 1 < \text{Suc } n$   
**by** *(metis Suc-1 Suc-eq-plus1 add.commute add-diff-cancel-left' add-less-mono1 grOI*  
*interv(1) interv(2) le-add-diff-inverse2 less-Suc-eq not-le power-0 power-one-right*  
*power-strict-increasing-iff)*  
**show** *?thesis*  
**apply** *(subst luby-sequence-core-not-two-power-minus-one[OF interv])*  
**using** *IH n* **by** *auto*

```

    qed
  qed
end

locale luby-sequence-restart =
  luby-sequence ur +
  cdclW-ops trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
  add-init-clss
  add-learned-clss remove-clss update-backtrack-lvl update-conflicting init-state
  restart-state
for
  ur :: nat and
  trail :: 'st  $\Rightarrow$  ('v::linorder, nat, 'v clause) marked-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 conflicting-clause and
  cons-trail :: ('v, nat, 'v clause) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-init-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  add-learned-clss remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause conflicting-clause  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v::linorder clauses  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st
begin

sublocale cdclW-ops-restart - - - - - luby-sequence
  apply unfold-locales
  using bounded-luby-sequence by blast

end

end
theory CDCL-W-Incremental
imports CDCL-W-Termination
begin

```

## 20 Incremental SAT solving

```

context cdclW-ops
begin

```

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl<sub>W</sub>-all-struct-inv*

```

definition cdclW-stgy-invariant where
  cdclW-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 cdclW-stgy-cdclW-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  $T$   
**unfolding**  $cdcl_W$ -stgy-invariant-def  $cdcl_W$ -all-struct-inv-def **apply** *standard*  
**apply** (rule  $cdcl_W$ -stgy-ex-lit-of-max-level[*of S*])  
**using** *assms* **unfolding**  $cdcl_W$ -stgy-invariant-def  $cdcl_W$ -all-struct-inv-def **apply** *auto*[7]  
**apply** *standard*  
**using**  $cdcl_W$   $cdcl_W$ -stgy-not-non-negated-init-clss **apply** *blast*  
**apply** *standard*  
**apply** (rule  $cdcl_W$ -stgy-no-smaller-conflict-inv)  
**using** *assms* **unfolding**  $cdcl_W$ -stgy-invariant-def  $cdcl_W$ -all-struct-inv-def **apply** *auto*[4]  
**using**  $cdcl_W$   $cdcl_W$ -stgy-not-non-negated-init-clss **by** *auto*

**lemma**  $rtranclp$ - $cdcl_W$ -stgy- $cdcl_W$ -stgy-invariant:  
**assumes**  
 $cdcl_W$ :  $cdcl_W$ -stgy\*\*  $S$   $T$  and  
 $inv$ -s:  $cdcl_W$ -stgy-invariant  $S$  and  
 $inv$ :  $cdcl_W$ -all-struct-inv  $S$   
**shows**  
 $cdcl_W$ -stgy-invariant  $T$   
**using** *assms* **apply** (*induction*)  
**apply** *simp*  
**using**  $cdcl_W$ -stgy- $cdcl_W$ -stgy-invariant  $rtranclp$ - $cdcl_W$ -all-struct-inv-inv  
 $rtranclp$ - $cdcl_W$ -stgy- $rtranclp$ - $cdcl_W$  **by** *blast*

**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  $C \ []$   $S = S \mid$   
 $cut$ -trail-wrt-clause  $C$  ( $Marked$   $L - \#$   $M$ )  $S =$   
 (if  $-L \in \#$   $C$  then  $S$   
   else  $cut$ -trail-wrt-clause  $C$   $M$  ( $decr$ -bt-lvl ( $tl$ -trail  $S$ )))  $\mid$   
 $cut$ -trail-wrt-clause  $C$  ( $Propagated$   $L - \#$   $M$ )  $S =$   
 (if  $-L \in \#$   $C$  then  $S$   
   else  $cut$ -trail-wrt-clause  $C$   $M$  ( $tl$ -trail  $S$ ))

**definition**  $add$ -new-clause-and-update :: ' $v$  literal multiset  $\Rightarrow$  ' $st \Rightarrow$  ' $st$  **where**  
 $add$ -new-clause-and-update  $C$   $S =$   
 (if trail  $S \models_{as}$   $CNot$   $C$   
   then  $update$ -conflicting ( $C$ -Clause  $C$ ) ( $add$ -init-cls  $C$  ( $cut$ -trail-wrt-clause  $C$  (trail  $S$ )  $S$ ))  
   else  $add$ -init-cls  $C$   $S$ )

**thm**  $cut$ -trail-wrt-clause.induct

**lemma**  $init$ -clss- $cut$ -trail-wrt-clause[*simp*]:  
 $init$ -clss ( $cut$ -trail-wrt-clause  $C$   $M$   $S$ ) =  $init$ -clss  $S$   
**by** (*induction rule: cut-trail-wrt-clause.induct*) *auto*

**lemma**  $learned$ -clss- $cut$ -trail-wrt-clause[*simp*]:  
 $learned$ -clss ( $cut$ -trail-wrt-clause  $C$   $M$   $S$ ) =  $learned$ -clss  $S$   
**by** (*induction rule: cut-trail-wrt-clause.induct*) *auto*

```

lemma conflicting-clss-cut-trail-wrt-clause[simp]:
  conflicting (cut-trail-wrt-clause C M S) = conflicting S
  by (induction rule: cut-trail-wrt-clause.induct) auto

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 (induction trail S arbitrary:S rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail S)] and M = this(2)[symmetric]
  then show ?case using Cons-eq-appendI by fastforce+
next
  case (proped L l M) note IH = this(1)[of (tl-trail S)] and M = this(2)[symmetric]
  then show ?case using Cons-eq-appendI by fastforce+
qed

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 –
  obtain M where
    M: trail T = M @ trail (cut-trail-wrt-clause C (trail T) T)
  using trail-cut-trail-wrt-clause[of T C] by auto
  show ?thesis
  using n-d unfolding arg-cong[OF M, of no-dup] by auto
qed

lemma cut-trail-wrt-clause-backtrack-lvl-length-marked:
  assumes
    backtrack-lvl T = length (get-all-levels-of-marked (trail T))
  shows
    backtrack-lvl (cut-trail-wrt-clause C (trail T) T) =
      length (get-all-levels-of-marked (trail (cut-trail-wrt-clause C (trail T) T)))
  using assms
proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
    and bt = this(3)
  then show ?case by auto
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt = this(3)
  then show ?case by auto
qed

lemma cut-trail-wrt-clause-get-all-levels-of-marked:
  assumes get-all-levels-of-marked (trail T) = rev [Suc 0..<
    Suc (length (get-all-levels-of-marked (trail T)))]
  shows
    get-all-levels-of-marked (trail ((cut-trail-wrt-clause C (trail T) T))) = rev [Suc 0..<
      Suc (length (get-all-levels-of-marked (trail ((cut-trail-wrt-clause C (trail T) T)))))]
  using assms

```



```

proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
  and bt = this(3)
  then show ?case by (cases count C L = 0) auto
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt = this(3)
  then show ?case by (cases count C L = 0) auto
qed

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
  using assms
proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
  and bt = this(3)

  then show ?case apply (cases count C (-L) = 0)
  apply (auto simp: true-annots-true-cls)
  by (smt CNot-def One-nat-def count-single diff-Suc-1 in-CNot-uminus less-numeral-extra(4)
    marked.prem1 marked-lit.sel(1) mem-Collect-eq true-annot-def true-annot-lit-of-notin-skip
    true-annots-def true-clss-def zero-less-diff)
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt = this(3)
  then show ?case

    apply (cases count C (-L) = 0)
    apply (auto simp: true-annots-true-cls)
    by (smt CNot-def One-nat-def count-single diff-Suc-1 in-CNot-uminus less-numeral-extra(4)
      proped.prem1 marked-lit.sel(2) mem-Collect-eq true-annot-def true-annot-lit-of-notin-skip
      true-annots-def true-clss-def zero-less-diff)
qed

lemma cut-trail-wrt-clause-hd-trail-in-or-empty-trail:
   $((\forall L \in \#C. -L \notin \text{ lits-of } (\text{trail } T)) \wedge \text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) = [])$ 
   $\vee (-\text{lit-of } (\text{hd } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))) \in \# C$ 
   $\wedge \text{length } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \geq 1$ 
  using assms
proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
  then show ?case by simp force
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric]
  then show ?case by simp force
qed

```

We can fully run  $cdcl_W$ -s or add a clause. Remark that we use  $cdcl_W$ -s to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict  $C$  if possible.

**inductive** *incremental-cdcl<sub>W</sub>* :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool **for**  $S$  **where**

*add-conflict*:

$trail\ S \models_{asm} init-clss\ S \implies distinct-mset\ C \implies conflicting\ S = C-True \implies$   
 $trail\ S \models_{as} CNot\ C \implies$   
*full cdcl<sub>W</sub>-stgy*  
 $(update-conflicting\ (C-Clause\ C)\ (add-init-clss\ C\ (cut-trail-wrt-clause\ C\ (trail\ S)\ S)))\ T \implies$   
*incremental-cdcl<sub>W</sub>*  $S\ T \mid$

*add-no-conflict*:

$trail\ S \models_{asm} init-clss\ S \implies distinct-mset\ C \implies conflicting\ S = C-True \implies$   
 $\neg trail\ S \models_{as} CNot\ C \implies$   
*full cdcl<sub>W</sub>-stgy*  $(add-init-clss\ C\ S)\ T \implies$   
*incremental-cdcl<sub>W</sub>*  $S\ T$

**inductive** *add-learned-clss* :: 'st  $\Rightarrow$  'v clauses  $\Rightarrow$  'st  $\Rightarrow$  bool **for**  $S$  :: 'st **where**

*add-learned-clss-nil*: *add-learned-clss*  $\{\#\}$   $S \mid$

*add-learned-clss-plus*:

*add-learned-clss*  $S\ A\ T \implies add-learned-clss\ S\ (\{\#x\# \} + A)\ (add-learned-clss\ x\ T)$

**declare** *add-learned-clss.intros*[intro]

**lemma** *Ex-add-learned-clss*:

$\exists T. add-learned-clss\ S\ A\ T$

**by** (*induction*  $A$  *arbitrary*:  $S$  *rule*: *multiset-induct*) (*auto simp*: *union-commute*[of -  $\{\#-\#\}$ ])

**lemma** *add-learned-clss-trail*:

**assumes** *add-learned-clss*  $S\ U\ T$  **and** *no-dup* ( $trail\ S$ )

**shows**  $trail\ T = trail\ S$

**using** *assms* **by** (*induction rule*: *add-learned-clss.induct*) (*simp-all add*: *ac-simps*)

**lemma** *add-learned-clss-learned-clss*:

**assumes** *add-learned-clss*  $S\ U\ T$  **and** *no-dup* ( $trail\ S$ )

**shows** *learned-clss*  $T = U + learned-clss\ S$

**using** *assms* **by** (*induction rule*: *add-learned-clss.induct*)

(*auto simp*: *ac-simps dest*: *add-learned-clss-trail*)

**lemma** *add-learned-clss-init-clss*:

**assumes** *add-learned-clss*  $S\ U\ T$  **and** *no-dup* ( $trail\ S$ )

**shows** *init-clss*  $T = init-clss\ S$

**using** *assms* **by** (*induction rule*: *add-learned-clss.induct*)

(*auto simp*: *ac-simps dest*: *add-learned-clss-trail*)

**lemma** *add-learned-clss-conflicting*:

**assumes** *add-learned-clss*  $S\ U\ T$  **and** *no-dup* ( $trail\ S$ )

**shows** *conflicting*  $T = conflicting\ S$

**using** *assms* **by** (*induction rule*: *add-learned-clss.induct*)

(*auto simp*: *ac-simps dest*: *add-learned-clss-trail*)

**lemma** *add-learned-clss-backtrack-lvl*:

**assumes** *add-learned-clss*  $S\ U\ T$  **and** *no-dup* ( $trail\ S$ )

**shows** *backtrack-lvl*  $T = backtrack-lvl\ S$

**using** *assms* **by** (*induction rule*: *add-learned-clss.induct*)

(*auto simp*: *ac-simps dest*: *add-learned-clss-trail*)

**lemma** *add-learned-clss-init-state-empty*[*dest*!]:

*add-learned-clss* (*init-state* *N*) {#} *T*  $\implies T = \text{init-state } N$   
**by** (*cases rule*: *add-learned-clss.cases*) (*auto simp*: *add-learned-clss.cases*)

For multiset larger than 1 element, there is no way to know in which order the clauses are added.  
 But contrary to a definition *fold-mset*, there is an element.

**lemma** *add-learned-clss-init-state-single*[*dest!*]:  
*add-learned-clss* (*init-state* *N*) {#*C*#} *T*  $\implies T = \text{add-learned-clss } C \text{ (init-state } N)$   
**by** (*induction* {#*C*#} *T* *rule*: *add-learned-clss.induct*)  
(*auto simp*: *add-learned-clss.cases* *ac-simps* *union-is-single* *split*: *split-if-asm*)

**thm** *rtranclp-cdcl<sub>W</sub>-stgy-no-smaller-confl-inv cdcl<sub>W</sub>-stgy-final-state-conclusive*

**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*  $\models_{\text{asm}} N$  **and**  
*tr-C*[*simp*]: *trail* *T*  $\models_{\text{as}} C \text{Not } C$  **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** –

**let** ?*T* = *update-conflicting* (*C-Clause* *C*) (*add-init-cls* *C* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*))

**obtain** *M* **where**

*M*: *trail* *T* = *M* @ *trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)  
**using** *trail-cut-trail-wrt-clause*[*of* *T* *C*] **by** *blast*

**have** *H*[*dest*]:  $\bigwedge x. x \in \text{lits-of } (\text{trail } (\text{cut-trail-wrt-clause } C \text{ (trail } T) \text{ } T)) \implies$   
 $x \in \text{lits-of } (\text{trail } T)$

**using** *inv-T* *arg-cong*[*OF* *M*, *of* *lits-of*] **by** *auto*

**have** *H'*[*dest*]:  $\bigwedge x. x \in \text{set } (\text{trail } (\text{cut-trail-wrt-clause } C \text{ (trail } T) \text{ } T)) \implies x \in \text{set } (\text{trail } T)$   
**using** *inv-T* *arg-cong*[*OF* *M*, *of* *set*] **by** *auto*

**have** *H-proped*:  $\bigwedge x. x \in \text{set } (\text{get-all-mark-of-propagated } (\text{trail } (\text{cut-trail-wrt-clause } C \text{ (trail } T) \text{ } T))) \implies$   
 $x \in \text{set } (\text{get-all-mark-of-propagated } (\text{trail } T))$

**using** *inv-T* *arg-cong*[*OF* *M*, *of* *get-all-mark-of-propagated*] **by** *auto*

**have** [*simp*]: *no-strange-atm* ?*T*

**using** *inv-T* **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* *no-strange-atm-def* *add-new-clause-and-update-def*  
*cdcl<sub>W</sub>-M-level-inv-def*  
**by** (*auto dest!*: *H* *H'*)

**have** *M-lev*: *cdcl<sub>W</sub>-M-level-inv* *T*

**using** *inv-T* **unfolding** *cdcl<sub>W</sub>-all-struct-inv-def* **by** *blast*

**then have** *no-dup* (*M* @ *trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*))

**unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **unfolding** *M*[*symmetric*] **by** *auto*

**then have** [*simp*]: *no-dup* (*trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*))  
**by** *auto*

**have** *consistent-interp* (*lits-of* (*M* @ *trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)))

**using** *M-lev* **unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **unfolding** *M*[*symmetric*] **by** *auto*

**then have** [*simp*]: *consistent-interp* (*lits-of* (*trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)))  
**unfolding** *consistent-interp-def* **by** *auto*

**have** [*simp*]: *cdcl<sub>W</sub>-M-level-inv* ?*T*

**unfolding** *cdcl<sub>W</sub>-M-level-inv-def* **apply** (*auto dest*: *H* *H'*)

*simp*: *M-lev* *cdcl<sub>W</sub>-M-level-inv-def* *cut-trail-wrt-clause-backtrack-lvl-length-marked*)

**using** *M-lev* *cut-trail-wrt-clause-get-all-levels-of-marked*[*of* *T* *C*]

**by** (*auto simp*: *cdcl<sub>W</sub>-M-level-inv-def* *cut-trail-wrt-clause-backtrack-lvl-length-marked*)

```

have [simp]:  $\bigwedge s. s \in \# \text{ learned-clss } T \implies \neg \text{tautology } s$ 
  using inv-T unfolding cdclW-all-struct-inv-def by auto

have distinct-cdclW-state T
  using inv-T unfolding cdclW-all-struct-inv-def by auto
then have [simp]: distinct-cdclW-state ?T
  unfolding distinct-cdclW-state-def by auto

have cdclW-conflicting T
  using inv-T unfolding cdclW-all-struct-inv-def by auto
have trail ?T  $\models_{as} CNot\ C$ 
  by (simp add: cut-trail-wrt-clause-CNot-trail)
then have [simp]: cdclW-conflicting ?T
  unfolding cdclW-conflicting-def apply simp
  by (metis M  $\langle cdcl_W\text{-conflicting } T \rangle$  append-assoc cdclW-conflicting-decomp(2))

have decomp-T: all-decomposition-implies-m (init-clss T) (get-all-marked-decomposition (trail T))
  using inv-T unfolding cdclW-all-struct-inv-def by auto
have all-decomposition-implies-m (init-clss ?T)
  (get-all-marked-decomposition (trail ?T))
  unfolding all-decomposition-implies-def
  proof clarify
    fix a b
    assume (a, b)  $\in$  set (get-all-marked-decomposition (trail ?T))
    from in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend[OF this]
    obtain b' where
      (a, b' @ b)  $\in$  set (get-all-marked-decomposition (trail T))
      using M by simp metis
    then have  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup \text{set-mset (init-clss ?T)}$ 
       $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } (b @ b')$ 
      using decomp-T unfolding all-decomposition-implies-def

    apply auto
    by (metis (no-types, lifting) case-prodD set-append sup commute true-clss-clss-insert-l)

    then show  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup \text{set-mset (init-clss ?T)}$ 
       $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } b$ 
      by (auto simp: image-Un)
  qed

have [simp]: cdclW-learned-clause ?T
  using inv-T unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def
  by (auto dest!: H-proped simp: clauses-def)
show ?thesis
  using  $\langle all\text{-decomposition-implies-m (init-clss ?T) (get-all-marked-decomposition (trail ?T)) \rangle$ 
  unfolding cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
qed

lemma cdclW-all-struct-inv-add-new-clause-and-update-cdclW-stgy-inv:
  assumes
    inv-s: cdclW-stgy-invariant T and
    inv: cdclW-all-struct-inv T and
    tr-T-N[simp]: trail T  $\models_{asm} N$  and

```

$tr-C[simp]: \text{trail } T \models_{as} CNot \ C \text{ and}$   
 $[simp]: \text{distinct-mset } C$   
**shows**  $cdcl_W\text{-stgy-invariant } (add\text{-new-clause-and-update } C \ T) \text{ (is } cdcl_W\text{-stgy-invariant } ?T')$   
**proof** –  
**have**  $cdcl_W\text{-all-struct-inv } ?T'$   
**using**  $cdcl_W\text{-all-struct-inv-add-new-clause-and-update-cdcl}_W\text{-all-struct-inv assms}$  **by** *blast*  
**then have**  
 $no\text{-dup-cut-}T[simp]: no\text{-dup } (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T)) \text{ and}$   
 $n\text{-d}[simp]: no\text{-dup } (trail \ T)$   
**using**  $cdcl_W\text{-M-level-inv-decomp}(2) \ cdcl_W\text{-all-struct-inv-def inv}$   
 $n\text{-dup-no-dup-trail-cut-trail-wrt-clause}$  **by** *blast+*  
**then have**  $trail \ (add\text{-new-clause-and-update } C \ T) \models_{as} CNot \ C$   
**by**  $(simp \ add: add\text{-new-clause-and-update-def cut-trail-wrt-clause-}CNot\text{-trail}$   
 $cdcl_W\text{-M-level-inv-def } cdcl_W\text{-all-struct-inv-def})$   
**obtain**  $MT$  **where**  
 $MT: trail \ T = MT \ @ \ trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T)$   
**using**  $trail\text{-cut-trail-wrt-clause}$  **by** *blast*  
**consider**  
 $(false) \ \forall L \in \#C. - L \notin \text{lits-of } (trail \ T) \text{ and } trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T) = []$   
 $| (not\text{-false}) - lit\text{-of } (hd \ (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T))) \in \# \ C \text{ and}$   
 $1 \leq \text{length } (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T))$   
**using**  $cut\text{-trail-wrt-clause-hd-trail-in-or-empty-trail}[of \ C \ T]$  **by** *auto*  
**then show**  $?thesis$   
**proof** *cases*  
**case** *false* **note**  $C = this(1)$  **and**  $empty\text{-tr} = this(2)$   
**then have**  $[simp]: C = \{\#\}$   
**by**  $(simp \ add: in\text{-}CNot\text{-implies-uminus}(2) \ multiset\text{-eqI})$   
**show**  $?thesis$   
**using**  $empty\text{-tr unfolding } cdcl_W\text{-stgy-invariant-def no-smaller-conflict-def}$   
 $cdcl_W\text{-all-struct-inv-def}$  **by**  $(auto \ simp: add\text{-new-clause-and-update-def})$   
**next**  
**case** *not-false* **note**  $C = this(1)$  **and**  $l = this(2)$   
**let**  $?L = - \ lit\text{-of } (hd \ (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T)))$   
**have**  $get\text{-all-levels-of-marked } (trail \ (add\text{-new-clause-and-update } C \ T)) =$   
 $rev \ [1..<1 + \text{length } (get\text{-all-levels-of-marked } (trail \ (add\text{-new-clause-and-update } C \ T)))]$   
**using**  $\langle cdcl_W\text{-all-struct-inv } ?T' \rangle$  **unfolding**  $cdcl_W\text{-all-struct-inv-def } cdcl_W\text{-M-level-inv-def}$   
**by** *blast*  
**moreover**  
**have**  $backtrack\text{-lvl } (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T) =$   
 $\text{length } (get\text{-all-levels-of-marked } (trail \ (add\text{-new-clause-and-update } C \ T)))$   
**using**  $\langle cdcl_W\text{-all-struct-inv } ?T' \rangle$  **unfolding**  $cdcl_W\text{-all-struct-inv-def } cdcl_W\text{-M-level-inv-def}$   
**by**  $(auto \ simp: add\text{-new-clause-and-update-def})$   
**moreover**  
**have**  $no\text{-dup } (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T))$   
**using**  $\langle cdcl_W\text{-all-struct-inv } ?T' \rangle$  **unfolding**  $cdcl_W\text{-all-struct-inv-def } cdcl_W\text{-M-level-inv-def}$   
**by**  $(auto \ simp: add\text{-new-clause-and-update-def})$   
**then have**  $atm\text{-of } ?L \notin atm\text{-of } ' \text{lits-of } (tl \ (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T)))$   
**apply**  $(cases \ trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T))$   
**apply**  $(auto)$   
**using**  $Marked\text{-Propagated-in-iff-in-lits-of defined-lit-map}$  **by** *blast*  
  
**ultimately have**  $L: get\text{-level } (-?L) \ (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T))$   
 $= \text{length } (get\text{-all-levels-of-marked } (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T)))$   
**using**  $get\text{-level-get-rev-level-get-all-levels-of-marked}[OF$   
 $\langle atm\text{-of } ?L \notin atm\text{-of } ' \text{lits-of } (tl \ (trail \ (cut\text{-trail-wrt-clause } C \ (trail \ T) \ T))) \rangle,$

```

of [hd (trail (cut-trail-wrt-clause C (trail T) T))]]

apply (cases trail (add-init-cls C (cut-trail-wrt-clause C (trail T) T));
  cases hd (trail (cut-trail-wrt-clause C (trail T) T)))
using l by (auto split: split-if-asm
  simp: rev-swap[symmetric] add-new-clause-and-update-def
  simp del:)

have L': length (get-all-levels-of-marked (trail (cut-trail-wrt-clause C (trail T) T)))
  = backtrack-lvl (cut-trail-wrt-clause C (trail T) T)
using ⟨cdclW-all-struct-inv ?T'⟩ unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
by (auto simp: add-new-clause-and-update-def)

have [simp]: no-smaller-confl (update-conflicting (C-Clause C)
  (add-init-cls C (cut-trail-wrt-clause C (trail T) T)))
unfolding no-smaller-confl-def
proof (clarify, goal-cases)
  case (1 M K i M' D)
  then consider
    (DC) D = C
  | (D-T) D ∈ # clauses T
  by (auto simp: clauses-def split: split-if-asm)
then show False
proof cases
  case D-T
  have no-smaller-confl T
    using inv-s unfolding cdclW-stgy-invariant-def by auto
  have (MT @ M') @ Marked K i # M = trail T
    using MT 1(1) by auto
  thus False using D-T ⟨no-smaller-confl T⟩ 1(3) unfolding no-smaller-confl-def by blast
next
  case DC note -[simp] = this
  then have atm-of (−?L) ∈ atm-of ' (lits-of M)
    using 1(3) C in-CNot-implies-uminus(2) by blast
  moreover
    have lit-of (hd (M' @ Marked K i # [])) = −?L
      using l 1(1)[symmetric] inv
    by (cases trail (add-init-cls C (cut-trail-wrt-clause C (trail T) T)))
      (auto dest!: arg-cong[of - # - - hd] simp: hd-append cdclW-all-struct-inv-def
        cdclW-M-level-inv-def)
    from arg-cong[OF this, of atm-of]
  have atm-of (−?L) ∈ atm-of ' (lits-of (M' @ Marked K i # []))
    by (cases (M' @ Marked K i # [])) auto
  moreover have no-dup (trail (cut-trail-wrt-clause C (trail T) T))
    using ⟨cdclW-all-struct-inv ?T'⟩ unfolding cdclW-all-struct-inv-def
    cdclW-M-level-inv-def by (auto simp: add-new-clause-and-update-def)
  ultimately show False
    unfolding 1(1)[symmetric, simplified]
    apply auto
    using Marked-Propagated-in-iff-in-lits-of defined-lit-map apply blast
    by (metis IntI Marked-Propagated-in-iff-in-lits-of defined-lit-map empty-iff)
qed
qed
show ?thesis using L L' C
  unfolding cdclW-stgy-invariant-def

```

**unfolding**  $cdcl_W\text{-all-struct-inv-def}$  **by** (*auto simp: add-new-clause-and-update-def*)  
**qed**  
**qed**

**lemma** *full-cdcl<sub>W</sub>-stgy-inv-normal-form*:  
**assumes**  
*full*:  $full\ cdcl_W\text{-stgy}\ S\ T$  **and**  
*inv-s*:  $cdcl_W\text{-stgy-inv}$   $S$  **and**  
*inv*:  $cdcl_W\text{-all-struct-inv}\ S$   
**shows**  $conflicting\ T = C\text{-Clause}\ \{\#\} \wedge unsatisfiable\ (set\text{-mset}\ (init\text{-clss}\ S))$   
 $\vee conflicting\ T = C\text{-True} \wedge trail\ T \models_{asm}\ init\text{-clss}\ S \wedge satisfiable\ (set\text{-mset}\ (init\text{-clss}\ S))$

**proof** –  
**have**  $no\text{-step}\ cdcl_W\text{-stgy}\ T$   
**using** *full* **unfolding** *full-def* **by** *blast*  
**moreover have**  $cdcl_W\text{-all-struct-inv}\ T$  **and**  $inv\text{-s}$ :  $cdcl_W\text{-stgy-inv}$   $T$   
**apply** (*metis*  $cdcl_W\text{-ops.rtranclp-cdcl}_W\text{-stgy-rtranclp-cdcl}_W\ cdcl_W\text{-ops-axioms}\ full\ full\text{-def}\ inv\ rtranclp\text{-cdcl}_W\text{-all-struct-inv-inv}$ )  
**by** (*metis*  $full\ full\text{-def}\ inv\ inv\text{-s}\ rtranclp\text{-cdcl}_W\text{-stgy-cdcl}_W\text{-stgy-inv}$ )  
**ultimately have**  $conflicting\ T = C\text{-Clause}\ \{\#\} \wedge unsatisfiable\ (set\text{-mset}\ (init\text{-clss}\ T))$   
 $\vee conflicting\ T = C\text{-True} \wedge trail\ T \models_{asm}\ init\text{-clss}\ T$   
**using**  $cdcl_W\text{-stgy-final-state-conclusive[of}\ T]\ full$   
**unfolding**  $cdcl_W\text{-all-struct-inv-def}\ cdcl_W\text{-stgy-inv-def}\ full\text{-def}$  **by** *fast*  
**moreover have**  $consistent\text{-interp}\ (lits\text{-of}\ (trail\ T))$   
**using**  $\langle cdcl_W\text{-all-struct-inv}\ T \rangle$  **unfolding**  $cdcl_W\text{-all-struct-inv-def}\ cdcl_W\text{-M-level-inv-def}$   
**by** *auto*  
**moreover have**  $init\text{-clss}\ S = init\text{-clss}\ T$   
**using** *inv* **unfolding**  $cdcl_W\text{-all-struct-inv-def}$   
**by** (*metis*  $rtranclp\text{-cdcl}_W\text{-stgy-no-more-init-clss}\ full\ full\text{-def}$ )  
**ultimately show** *?thesis*  
**by** (*metis*  $satisfiable\text{-carac}'\ true\text{-annot-def}\ true\text{-annots-def}\ true\text{-clss-def}$ )  
**qed**

**lemma** *incremental-cdcl<sub>W</sub>-inv*:  
**assumes**  
*inc*:  $incremental\text{-cdcl}_W\ S\ T$  **and**  
*inv*:  $cdcl_W\text{-all-struct-inv}\ S$  **and**  
*s-inv*:  $cdcl_W\text{-stgy-inv}$   $S$   
**shows**  
 $cdcl_W\text{-all-struct-inv}\ T$  **and**  
 $cdcl_W\text{-stgy-inv}$   $T$   
**using** *inc*

**proof** (*induction*)  
**case** ( $add\text{-confl}\ C\ T$ )  
**let**  $?T = (update\text{-conflicting}\ (C\text{-Clause}\ C)\ (add\text{-init-cl}\ C\ (cut\text{-trail-wrt-clause}\ C\ (trail\ S)\ S)))$   
**have**  $cdcl_W\text{-all-struct-inv}\ ?T$  **and**  $inv\text{-s-T}$ :  $cdcl_W\text{-stgy-inv}$   $?T$   
**using**  $add\text{-confl.hyps}(1,2,4)\ add\text{-new-clause-and-update-def}\ cdcl_W\text{-all-struct-inv-add-new-clause-and-update-cdcl}_W\text{-all-struct-inv}\ inv$  **apply** *auto*[1]  
**using**  $add\text{-confl.hyps}(1,2,4)\ add\text{-new-clause-and-update-def}\ cdcl_W\text{-all-struct-inv-add-new-clause-and-update-cdcl}_W\text{-stgy-inv}\ inv\ s\text{-inv}$  **by** *auto*  
**case 1 show** *?case*  
**by** (*metis*  $add\text{-confl.hyps}(1,2,4,5)\ add\text{-new-clause-and-update-def}\ cdcl_W\text{-all-struct-inv-add-new-clause-and-update-cdcl}_W\text{-all-struct-inv}\ rtranclp\text{-cdcl}_W\text{-all-struct-inv-inv}\ rtranclp\text{-cdcl}_W\text{-stgy-rtranclp-cdcl}_W\ full\text{-def}\ inv$ )  
**case 2 show** *?case*

```

by (metis inv-s-T add-confl.hyps(1,2,4,5) add-new-clause-and-update-def
    cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv full-def inv
    rtranclp-cdclW-stgy-cdclW-stgy-invariant)
next
case (add-no-confl C T)
case 1
have cdclW-all-struct-inv (add-init-cls C S)
  using inv ⟨distinct-mset C⟩ unfolding cdclW-all-struct-inv-def no-strange-atm-def
  cdclW-M-level-inv-def distinct-cdclW-state-def cdclW-conflicting-def cdclW-learned-clause-def
  by (auto simp: all-decomposition-implies-insert-single clauses-def)
then show ?case
  using add-no-confl(5) unfolding full-def by (auto intro: rtranclp-cdclW-stgy-cdclW-all-struct-inv)
case 2 have cdclW-stgy-invariant (add-init-cls C S)
  using s-inv ⟨¬ trail S ⊨as CNot C⟩ inv unfolding cdclW-stgy-invariant-def no-smaller-confl-def
  eq-commute[of - trail -] cdclW-M-level-inv-def cdclW-all-struct-inv-def
  by (auto simp: true-annots-true-cls-def-iff-negation-in-model clauses-def split: split-if-asm)
then show ?case
  by (metis ⟨cdclW-all-struct-inv (add-init-cls C S)⟩ add-no-confl.hyps(5) full-def
      rtranclp-cdclW-stgy-cdclW-stgy-invariant)
qed

```

**lemma** *rtranclp-incremental-cdcl<sub>W</sub>-inv:*

```

assumes
  inc: incremental-cdclW** S T and
  inv: cdclW-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows
  cdclW-all-struct-inv T and
  cdclW-stgy-invariant T
  using inc apply induction
  using inv apply simp
  using s-inv apply simp
using incremental-cdclW-inv by blast+

```

**lemma** *incremental-conclusive-state:*

```

assumes
  inc: incremental-cdclW S T and
  inv: cdclW-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows conflicting T = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss T))
  ∨ conflicting T = C-True ∧ trail T ⊨asm init-clss T ∧ satisfiable (set-mset (init-clss T))
using inc apply induction

```

```

apply (metis Nitpick.rtranclp-unfold add-confl full-cdclW-stgy-inv-normal-form full-def
    incremental-cdclW-inv(1) incremental-cdclW-inv(2) inv s-inv)
by (metis (full-types) rtranclp-unfold add-no-confl full-cdclW-stgy-inv-normal-form
    full-def incremental-cdclW-inv(1) incremental-cdclW-inv(2) inv s-inv)

```

**lemma** *tranclp-incremental-correct:*

```

assumes
  inc: incremental-cdclW++ S T and
  inv: cdclW-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows conflicting T = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss T))
  ∨ conflicting T = C-True ∧ trail T ⊨asm init-clss T ∧ satisfiable (set-mset (init-clss T))

```



**using** *inc apply induction*  
**using** *assms incremental-conclusive-state apply blast*  
**by** (*meson incremental-conclusive-state inv rtranclp-incremental-cdcl<sub>W</sub>-inv s-inv*  
*tranclp-into-rtranclp*)

**lemma** *blocked-induction-with-marked*:

**assumes**

*n-d*: *no-dup* ( $L \# M$ ) **and**

*nil*:  $P []$  **and**

*append*:  $\bigwedge M L M'. P M \implies \text{is-marked } L \implies \forall m \in \text{set } M'. \neg \text{is-marked } m \implies \text{no-dup } (L \# M' @ M) \implies$

$P (L \# M' @ M)$  **and**

*L*: *is-marked*  $L$

**shows**

$P (L \# M)$

**using** *n-d L*

**proof** (*induction card*  $\{L' \in \text{set } M. \text{is-marked } L'\}$  *arbitrary*:  $L M$ )

**case** 0 **note**  $n = \text{this}(1)$  **and**  $n-d = \text{this}(2)$  **and**  $L = \text{this}(3)$

**then have**  $\forall m \in \text{set } M. \neg \text{is-marked } m$  **by** *auto*

**then show** ?*case* **using** *append[of [] L M] L nil n-d* **by** *auto*

**next**

**case** (*Suc*  $n$ ) **note**  $IH = \text{this}(1)$  **and**  $n = \text{this}(2)$  **and**  $n-d = \text{this}(3)$  **and**  $L = \text{this}(4)$

**have**  $\exists L' \in \text{set } M. \text{is-marked } L'$

**proof** (*rule ccontr*)

**assume**  $\neg ?thesis$

**then have**  $H: \{L' \in \text{set } M. \text{is-marked } L'\} = \{\}$

**by** *auto*

**show** *False* **using** *n unfolding H* **by** *auto*

**qed**

**then obtain**  $L' M' M''$  **where**

$M: M = M' @ L' \# M''$  **and**

$L': \text{is-marked } L'$  **and**

$nm: \forall m \in \text{set } M'. \neg \text{is-marked } m$

**by** (*auto elim!: split-list-first-propE*)

**have**  $\text{Suc } n = \text{card } \{L' \in \text{set } M. \text{is-marked } L'\}$

**using** *n* .

**moreover have**  $\{L' \in \text{set } M. \text{is-marked } L'\} = \{L'\} \cup \{L' \in \text{set } M''. \text{is-marked } L'\}$

**using**  $nm L' n-d$  **unfolding**  $M$  **by** *auto*

**moreover have**  $L' \notin \{L' \in \text{set } M''. \text{is-marked } L'\}$

**using**  $n-d$  **unfolding**  $M$  **by** *auto*

**ultimately have**  $n = \text{card } \{L'' \in \text{set } M''. \text{is-marked } L''\}$

**using**  $n L'$  **by** *auto*

**then have**  $P (L' \# M'')$  **using**  $IH L' n-d M$  **by** *auto*

**then show** ?*case* **using** *append[of L' # M'' L M] nm L n-d unfolding M* **by** *blast*

**qed**

**lemma** *trail-bloc-induction*:

**assumes**

*n-d*: *no-dup*  $M$  **and**

*nil*:  $P []$  **and**

*append*:  $\bigwedge M L M'. P M \implies \text{is-marked } L \implies \forall m \in \text{set } M'. \neg \text{is-marked } m \implies \text{no-dup } (L \# M' @ M) \implies$

$P (L \# M' @ M)$  **and**

*append-nm*:  $\bigwedge M' M''. P M' \implies M = M'' @ M' \implies \forall m \in \text{set } M''. \neg \text{is-marked } m \implies P M$

**shows**

```

    P M
proof (cases {L' ∈ set M. is-marked L'} = {})
  case True
    then show ?thesis using append-nm[of [] M] nil by auto
next
  case False
    then have ∃ L' ∈ set M. is-marked L'
      by auto
    then obtain L' M' M'' where
      M: M = M' @ L' # M'' and
      L': is-marked L' and
      nm: ∀ m ∈ set M'. ¬is-marked m
      by (auto elim!: split-list-first-propE)
    have P (L' # M'')
      apply (rule blocked-induction-with-marked)
        using n-d unfolding M apply simp
        using nil apply simp
        using append apply simp
        using L' by auto
    then show ?thesis
      using append-nm[of - M'] nm unfolding M by simp
qed

inductive Tcons :: ('v, nat, 'v clause) marked-lits ⇒ ('v, nat, 'v clause) marked-lits ⇒ bool
  for M :: ('v, nat, 'v clause) marked-lits where
    Tcons M [] |
    Tcons M M' ⇒ M = M'' @ M' ⇒ (∀ m ∈ set M''. ¬is-marked m) ⇒ Tcons M (M'' @ M') |
    Tcons M M' ⇒ is-marked L ⇒ M = M''' @ L # M'' @ M' ⇒ (∀ m ∈ set M''. ¬is-marked m) ⇒
      Tcons M (L # M'' @ M')

lemma Tcons-same-end: Tcons M M' ⇒ ∃ M''. M = M'' @ M'
  by (induction rule: Tcons.induct) auto

end

end

theory CDCL-Two-Watched-Literals
imports CDCL-WNOT
begin

Only the 2-watched literals have to be verified here: the backtrack level and the trail can remain
separate.

datatype 'v twl-clause =
  TWL-Clause (watched: 'v clause) (unwatched: 'v clause)

abbreviation raw-clause :: 'v twl-clause ⇒ 'v clause where
  raw-clause C ≡ watched C + unwatched C

datatype ('v, 'vl, 'mark) twl-state =
  TWL-State (trail: ('v, 'vl, 'mark) marked-lits) (init-clss: 'v twl-clause multiset)
    (learned-clss: 'v twl-clause multiset) (backtrack-lvl: 'vl)
    (conflicting: 'v clause conflicting-clause)

```

**abbreviation** *raw-init-clss* **where**

*raw-init-clss*  $S \equiv \text{image-mset } \text{raw-clause } (\text{init-clss } S)$

**abbreviation** *raw-learned-clsss* **where**

*raw-learned-clsss*  $S \equiv \text{image-mset } \text{raw-clause } (\text{learned-clss } S)$

**abbreviation** *clauses* **where**

*clauses*  $S \equiv \text{init-clss } S + \text{learned-clss } S$

**definition**

*candidates-propagate*  $:: ('v, 'lv, 'mark) \text{ twl-state} \Rightarrow ('v \text{ literal} \times 'v \text{ clause}) \text{ set}$

**where**

*candidates-propagate*  $S =$

$\{(L, \text{raw-clause } C) \mid L \in C.$

$C \in \# \text{ clauses } S \wedge \text{watched } C - \text{mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S)) = \{\#L\# \} \wedge$

$\text{undefined-lit } (\text{trail } S) L\}$

**definition** *candidates-conflict*  $:: ('v, 'lv, 'mark) \text{ twl-state} \Rightarrow 'v \text{ clause set}$  **where**

*candidates-conflict*  $S =$

$\{\text{raw-clause } C \mid C. C \in \# \text{ clauses } S \wedge \text{watched } C \subseteq \# \text{ mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S))\}$

**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*:

$a \in \text{set } l \implies l ! (\text{index } l a) = a$

**by** (*induction*  $l$ ) *auto*

We need the following property: 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* it does not get swap with a watched literal  $L'$  such that  $-L'$  is in the trail.

**primrec** *watched-decided-most-recently*  $:: ('v, 'lv, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ twl-clause} \Rightarrow \text{bool}$

**where**

*watched-decided-most-recently*  $M (\text{TWL-Clause } W \text{ } UW) \longleftrightarrow$

$(\forall L' \in \# W. \forall L \in \# UW.$

$-L' \in \text{lits-of } M \longrightarrow -L \in \text{lits-of } M \longrightarrow$

$\text{index } (\text{map lit-of } M) (-L') \leq \text{index } (\text{map lit-of } M) (-L))$

**primrec** *wf-tw-cl*  $:: ('v, 'lv, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ twl-clause} \Rightarrow \text{bool}$  **where**

*wf-tw-cl*  $M (\text{TWL-Clause } W \text{ } UW) \longleftrightarrow$

$\text{distinct-mset } W \wedge \text{size } W \leq 2 \wedge (\text{size } W < 2 \longrightarrow \text{set-mset } UW \subseteq \text{set-mset } W) \wedge$

$(\forall L \in \# W. -L \in \text{lits-of } M \longrightarrow (\forall L' \in \# UW. L' \notin W \longrightarrow -L' \in \text{lits-of } M)) \wedge$

*watched-decided-most-recently*  $M (\text{TWL-Clause } W \text{ } UW)$

**lemma**  $-L \in \text{lits-of } M \implies \{i. \text{map lit-of } M ! i = -L\} \neq \{\}$

**unfolding** *set-map-lit-of-lits-of* [*symmetric*] *set-conv-nth*

**by** (*smt Collect-empty-eq mem-Collect-eq*)

**lemma** *size-mset-2*:  $\text{size } x1 = 2 \longleftrightarrow (\exists a b. x1 = \{\#a, b\# \})$

**by** (*metis* (*no-types*, *hide-lams*) *Suc-eq-plus1 one-add-one size-1-singleton-mset*

*size-Diff-singleton size-Suc-Diff1 size-eq-Suc-imp-eq-union size-single union-single-eq-diff*

*union-single-eq-member*)

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

**unfolding** *distinct-mset-def* **by** *auto*

does not hold when all there are multiple conflicts in a clause.

```

lemma wf-twl-cls-wf-twl-cls-tl:
  assumes wf: wf-twl-cls M C and n-d: no-dup M
  shows wf-twl-cls (tl M) C
proof (cases M)
  case Nil
  then show ?thesis using wf
  by (cases C) (simp add: wf-twl-cls.simps[of tl -])
next
  case (Cons l M') note M = this(1)
  obtain W UW where C: C = TWL-Clause W UW
  by (cases C)
  { fix L L'
    assume
      LW: L ∈# W and
      LM: − L ∈ lits-of M' and
      L'UW: L' ∈# UW and
      count W L' = 0
    then have
      L'M: − L' ∈ lits-of M
      using wf by (auto simp: C M)
    have watched-decided-most-recently M C
      using wf by (auto simp: C)
    then have
      index (map lit-of M) (−L) ≤ index (map lit-of M) (−L')
      using LM L'M L'UW LW by (metis (mono-tags, lifting) C M bspec-mset insert-iff lits-of-cons
        watched-decided-most-recently.simps)
    then have − L' ∈ lits-of M'
      using ⟨count W L' = 0⟩ LW L'M by (auto simp: C M split: split-if-asm)
  }
moreover
  {
    fix L L' La
    assume
      L ∈# W and
      LM': − L ∈ lits-of M' and
      L' ∈# W and
      La ∈# UW and
      L'M: − L' ∈ lits-of M' and
      − La ∈ lits-of M'
    moreover
      have lit-of l ≠ − L'
      using n-d
      by (metis (no-types) L'M M Marked-Propagated-in-iff-in-lits-of defined-lit-map
        distinct.simps(2) list.simps(9) set-map)
    moreover have watched-decided-most-recently M C
      using wf by (auto simp: C)
    ultimately have index (map lit-of M') (− L') ≤ index (map lit-of M') (− La)
      by (auto simp: M C split: split-if-asm)
  }
moreover have distinct-mset W and size W ≤ 2 and (size W < 2 ⟶ set-mset UW ⊆ set-mset W)
  using wf by (auto simp: C M)
ultimately show ?thesis by (simp add: M C)
qed

```

```

definition wf-tw1-state :: ('v, 'wl, 'mark) tw1-state  $\Rightarrow$  bool where
  wf-tw1-state S  $\longleftrightarrow$  ( $\forall C \in \#$  clauses S. wf-tw1-cls (trail S) C)  $\wedge$  no-dup (trail S)

lemma wf-candidates-propagate-sound:
  assumes wf: wf-tw1-state S and
    cand: (L, C)  $\in$  candidates-propagate S
  shows trail S  $\models_{as}$  CNot (mset-set (set-mset C - {L}))  $\wedge$  undefined-lit (trail S) L
proof
  def M  $\equiv$  trail S
  def N  $\equiv$  init-clss S
  def U  $\equiv$  learned-clss S

  note MNU-defs [simp] = M-def N-def U-def

  obtain Cw where cw:
    C = raw-clause Cw
    Cw  $\in \#$  N + U
    watched Cw - mset-set (uminus ' lits-of M) = {#L#}
    undefined-lit M L
  using cand unfolding candidates-propagate-def MNU-defs by blast

  obtain W UW where cw-eq: Cw = TWL-Clause W UW
  by (case-tac Cw, blast)

  have l-w: L  $\in \#$  W
  by (metis Multiset.diff-le-self cw(3) cw-eq mset-leD multi-member-last tw1-clause.sel(1))

  have wf-c: wf-tw1-cls M Cw
  using wf (Cw  $\in \#$  N + U) unfolding wf-tw1-state-def by simp

  have w-nw:
    distinct-mset W
    size W < 2  $\implies$  set-mset UW  $\subseteq$  set-mset W
     $\bigwedge L L'. L \in \# W \implies -L \in$  lits-of M  $\implies L' \in \# UW \implies L' \notin \# W \implies -L' \in$  lits-of M
  using wf-c unfolding cw-eq by auto

  have  $\forall L' \in$  set-mset C - {L}.  $-L' \in$  lits-of M
  proof (cases size W < 2)
  case True
  moreover have size W  $\neq$  0
  using cw(3) cw-eq by auto
  ultimately have size W = 1
  by linarith
  then have w: W = {#L#}
  by (metis (no-types, lifting) Multiset.diff-le-self cw(3) cw-eq single-not-empty
    size-1-singleton-mset subset-mset.add-diff-inverse union-is-single tw1-clause.sel(1))
  from True have set-mset UW  $\subseteq$  set-mset W
  using w-nw(2) by blast
  then show ?thesis
  using w cw(1) cw-eq by auto
next
  case sz2: False
  show ?thesis
  proof

```

```

fix L'
assume l': L' ∈ set-mset C - {L}
have ex-la: ∃ La. La ≠ L ∧ La ∈# W
proof (cases W)
  case empty
  thus ?thesis
    using l-w by auto
next
  case lb: (add W' Lb)
  show ?thesis
  proof (cases W')
    case empty
    thus ?thesis
      using lb sz2 by simp
  next
    case lc: (add W'' Lc)
    thus ?thesis
      by (metis add-gr-0 count-union distinct-mset-single-add lb union-single-eq-member
        w-nw(1))
  qed
qed
then obtain La where la: La ≠ L La ∈# W
  by blast
then have La ∈# mset-set (uminus ' lits-of M)
  using cw(3)[unfolded cw-eq, simplified, folded M-def]
  by (metis count-diff count-single diff-zero not-gr0)
then have nla: -La ∈ lits-of M
  by auto
then show -L' ∈ lits-of M

proof -
  have f1: L' ∈ set-mset C
    using l' by blast
  have f2: L' ∉ {L}
    using l' by fastforce
  have ∧l L. - (l::'a literal) ∈ L ∨ l ∉ uminus ' L
    by force
  then have ∧l. - l ∈ lits-of M ∨ count {#L#} l = count (C - UW) l
    by (metis (no-types) add-diff-cancel-right' count-diff count-mset-set(3) cw(1) cw(3)
      cw-eq diff-zero twl-clause.sel(2))
  then show ?thesis
    by (smt comm-monoid-add-class.add-0 cw(1) cw-eq diff-union-cancelR ex-la f1 f2 insertCI
      less-numeral-extra(3) mem-set-mset-iff plus-multiset.rep-eq single.rep-eq
      twl-clause.sel(1) twl-clause.sel(2) w-nw(3))
  qed
qed
qed
then show trail S ⊨as CNot (mset-set (set-mset C - {L}))
  unfolding true-annots-def by auto

show undefined-lit (trail S) L
  using cw(4) M-def by blast
qed

```

**lemma** wf-candidates-propagate-complete:

```

assumes wf: wf-twl-state  $S$  and
  c-mem:  $C \in \#$  image-mset raw-clause (clauses  $S$ ) and
  l-mem:  $L \in \#$   $C$  and
  unsat: trail  $S \models_{as} CNot$  (mset-set (set-mset  $C - \{L\}$ )) and
  undef: undefined-lit (trail  $S$ )  $L$ 
shows  $(L, C) \in$  candidates-propagate  $S$ 
proof –
  def  $M \equiv$  trail  $S$ 
  def  $N \equiv$  init-clss  $S$ 
  def  $U \equiv$  learned-clss  $S$ 

  note  $MNU$ -defs [simp] =  $M$ -def  $N$ -def  $U$ -def

  obtain  $Cw$  where  $cw$ :  $C =$  raw-clause  $Cw$   $Cw \in \#$   $N + U$ 
    using c-mem by force

  obtain  $W$   $UW$  where  $cw$ -eq:  $Cw =$  TWL-Clause  $W$   $UW$ 
    by (case-tac  $Cw$ , blast)

  have wf-c: wf-twl-clss  $M$   $Cw$ 
    using wf  $cw(2)$  unfolding wf-twl-state-def by simp

  have w-nw:
    distinct-mset  $W$ 
    size  $W < 2 \implies$  set-mset  $UW \subseteq$  set-mset  $W$ 
     $\bigwedge L L'. L \in \# W \implies -L \in$  lits-of  $M \implies L' \in \# UW \implies L' \notin \# W \implies -L' \in$  lits-of  $M$ 
    using wf-c unfolding cw-eq by auto

  have unit-set: set-mset  $(W -$  mset-set (uminus ‘lits-of  $M$ )) =  $\{L\}$ 
  proof
    show set-mset  $(W -$  mset-set (uminus ‘lits-of  $M$ ))  $\subseteq \{L\}$ 
    proof
      fix  $L'$ 
      assume  $l'$ :  $L' \in$  set-mset  $(W -$  mset-set (uminus ‘lits-of  $M$ ))
      hence  $l'$ -mem-w:  $L' \in$  set-mset  $W$ 
      by auto
      have  $L' \notin$  uminus ‘lits-of  $M$ 
        using distinct-mem-diff-mset[OF w-nw(1)  $l'$ ] by simp
      then have  $\neg M \models_a \{\# - L'\# \}$ 
        using image-iff by fastforce
      moreover have  $L' \in \# C$ 
        using  $cw(1)$  cw-eq  $l'$ -mem-w by auto
      ultimately have  $L' = L$ 
        unfolding  $M$ -def by (metis unsat[unfolded CNot-def true-annots-def, simplified])
      then show  $L' \in \{L\}$ 
        by simp
    qed
  next
    show  $\{L\} \subseteq$  set-mset  $(W -$  mset-set (uminus ‘lits-of  $M$ ))
    proof clarify
      have  $L \in \# W$ 
      proof (cases  $W$ )
        case empty
        thus ?thesis
        using w-nw(2)  $cw(1)$  cw-eq l-mem by auto

```

```

next
  case (add W' La)
  thus ?thesis
  proof (cases La = L)
    case True
    thus ?thesis
      using add by simp
  next
    case False
    have  $\neg La \in \text{ lits-of } M$ 
      using False add cw(1) cw-eq unsat[unfolded CNot-def true-annots-def, simplified]
      by fastforce
    then show ?thesis
      by (metis M-def Marked-Propagated-in-iff-in-lits-of add add.left-neutral count-union
        cw(1) cw-eq grOI l-mem twl-clause.sel(1) twl-clause.sel(2) undef union-single-eq-member
        w-nw(3))
    qed
  qed
  moreover have  $L \notin \# \text{ mset-set } (\text{uminus } ' \text{ lits-of } M)$ 
    using Marked-Propagated-in-iff-in-lits-of undef by auto
  ultimately show  $L \in \text{ set-mset } (W - \text{ mset-set } (\text{uminus } ' \text{ lits-of } M))$ 
    by auto
  qed
  qed
  have unit:  $W - \text{ mset-set } (\text{uminus } ' \text{ lits-of } M) = \{\#L\# \}$ 
    by (metis distinct-mset-minus distinct-mset-set-mset-ident distinct-mset-singleton
      set-mset-single unit-set w-nw(1))

  show ?thesis
    unfolding candidates-propagate-def using unit undef cw cw-eq by fastforce
  qed

lemma wf-candidates-conflict-sound:
  assumes wf: wf-twl-state S and
    cand:  $C \in \text{ candidates-conflict } S$ 
  shows  $\text{trail } S \models_{as} \text{CNot } C \wedge C \in \# \text{ image-mset raw-clause } (\text{clauses } S)$ 
proof
  def M  $\equiv \text{trail } S$ 
  def N  $\equiv \text{init-clss } S$ 
  def U  $\equiv \text{learned-clss } S$ 

  note MNU-defs [simp] = M-def N-def U-def

  obtain Cw where cw:
    C = raw-clause Cw
    Cw  $\in \# N + U$ 
    watched Cw  $\subseteq \# \text{ mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S))$ 
    using cand[unfolded candidates-conflict-def, simplified] by auto

  obtain W UW where cw-eq: Cw = TWL-Clause W UW
    by (case-tac Cw, blast)

  have wf-c: wf-twl-clss M Cw
    using wf cw(2) unfolding wf-twl-state-def by simp

```



```

have w-nw:
  distinct-mset W
  size W < 2 ==> set-mset UW ⊆ set-mset W
  ∧ L L'. L ∈# W ==> -L ∈ lits-of M ==> L' ∈# UW ==> L' ∉# W ==> -L' ∈ lits-of M
  using wf-c unfolding cw-eq by auto

have ∀ L ∈# C. -L ∈ lits-of M
proof (cases W = {#})
  case True
  then have C = {#}
    using cw(1) cw-eq w-nw(2) by auto
  then show ?thesis
    by simp
next
  case False
  then obtain La where la: La ∈# W
    using multiset-eq-iff by force
  show ?thesis
  proof
    fix L
    assume l: L ∈# C
    show -L ∈ lits-of M
    proof (cases L ∈# W)
      case True
      thus ?thesis
        using cw(3) cw-eq by fastforce
    next
      case False
      thus ?thesis
        by (smt M-def l add-diff-cancel-left' count-diff cw(1) cw(3) la cw-eq
            diff-zero elem-mset-set finite-imageI finite-lits-of-def grOI imageE mset-leD
            uminus-of-uminus-id twl-clause.sel(1) twl-clause.sel(2) w-nw(3))
    qed
  qed
  qed
  then show trail S ⊨as CNot C
    unfolding CNot-def true-annots-def by auto

  show C ∈# image-mset raw-clause (clauses S)
    using cw by auto
qed

lemma wf-candidates-conflict-complete:
  assumes wf: wf-twI-state S and
    c-mem: C ∈# image-mset raw-clause (clauses S) and
    unsat: trail S ⊨as CNot C
  shows C ∈ candidates-conflict S
proof -
  def M ≡ trail S
  def N ≡ init-clss S
  def U ≡ learned-clss S

  note MNU-defs [simp] = M-def N-def U-def

  obtain Cw where cw: C = raw-clause Cw Cw ∈# N + U

```

```

using c-mem by force

obtain  $W \text{ } UW$  where  $cw\text{-eq}: Cw = \text{TWL-Clause } W \text{ } UW$ 
by (case-tac  $Cw$ , blast)

have  $wf\text{-c}: wf\text{-twl-cl}\text{ } M \text{ } Cw$ 
using  $wf \text{ } cw(2)$  unfolding  $wf\text{-twl-state-def}$  by simp

have  $w\text{-nw}$ :
  distinct-mset  $W$ 
  size  $W < 2 \implies \text{set-mset } UW \subseteq \text{set-mset } W$ 
   $\bigwedge L \text{ } L'. L \in \# \text{ } W \implies -L \in \text{lits-of } M \implies L' \in \# \text{ } UW \implies L' \notin \# \text{ } W \implies -L' \in \text{lits-of } M$ 
using  $wf\text{-c}$  unfolding  $cw\text{-eq}$  by auto

have  $\bigwedge L. L \in \# \text{ } C \implies -L \in \text{lits-of } M$ 
unfolding  $M\text{-def}$  using unsat[unfolded  $CNot\text{-def}$  true-annots-def, simplified] by blast
then have  $\text{set-mset } C \subseteq \text{uminus ' lits-of } M$ 
by (metis imageI mem-set-mset-iff subsetI uminus-of-uminus-id)
then have  $\text{set-mset } W \subseteq \text{uminus ' lits-of } M$ 
using  $cw(1)$   $cw\text{-eq}$  by auto
then have  $\text{subset}: W \subseteq \# \text{ mset-set (uminus ' lits-of } M)$ 
by (simp add: w-nw(1))

have  $W = \text{watched } Cw$ 
using  $cw\text{-eq}$   $\text{twl-clause.sel}(1)$  by simp
then show ?thesis
using  $MNU\text{-defs}$   $cw(1)$   $cw(2)$  subset candidates-conflict-def by blast
qed

typedef  $'v \text{ } wf\text{-twl} = \{S::('v, \text{nat}, 'v \text{ clause}) \text{ twl-state. } wf\text{-twl-state } S\}$ 
morphisms rough-state-of-tw $l$  twl-of-rough-state
proof –
  have  $\text{TWL-State } ([::('v, \text{nat}, 'v \text{ clause}) \text{ marked-lits})$ 
     $\{\#\} \{\#\} 0 \text{ } C\text{-True} \in \{S::('v, \text{nat}, 'v \text{ clause}) \text{ twl-state. } wf\text{-twl-state } S\}$ 
    by (auto simp: wf-tw $l\text{-state-def}$ )
  then show ?thesis by auto
qed

lemma  $wf\text{-twl-state-rough-state-of-tw}$  $l$ [simp]:  $wf\text{-twl-state (rough-state-of-tw}$  $l$   $S)$ 
using rough-state-of-tw $l$  by auto

abbreviation  $\text{candidates-conflict-tw}$  $l$  ::  $'v \text{ } wf\text{-twl} \Rightarrow 'v \text{ literal multiset set}$  where
 $\text{candidates-conflict-tw}$  $l$   $S \equiv \text{candidates-conflict (rough-state-of-tw}$  $l$   $S)$ 

abbreviation  $\text{candidates-propagate-tw}$  $l$  ::  $'v \text{ } wf\text{-twl} \Rightarrow ('v \text{ literal} \times 'v \text{ clause}) \text{ set}$  where
 $\text{candidates-propagate-tw}$  $l$   $S \equiv \text{candidates-propagate (rough-state-of-tw}$  $l$   $S)$ 

abbreviation  $\text{trail-tw}$  $l$  ::  $'a \text{ } wf\text{-twl} \Rightarrow ('a, \text{nat}, 'a \text{ literal multiset}) \text{ marked-lit list}$  where
 $\text{trail-tw}$  $l$   $S \equiv \text{trail (rough-state-of-tw}$  $l$   $S)$ 

abbreviation  $\text{clauses-tw}$  $l$  ::  $'a \text{ } wf\text{-twl} \Rightarrow 'a \text{ twl-clause multiset}$  where
 $\text{clauses-tw}$  $l$   $S \equiv \text{clauses (rough-state-of-tw}$  $l$   $S)$ 

abbreviation  $\text{init-clss-tw}$  $l$  where
 $\text{init-clss-tw}$  $l$   $S \equiv \text{image-mset raw-clause (init-clss (rough-state-of-tw}$  $l$   $S))$ 

```

**abbreviation** *learned-clss-twl* **where**

*learned-clss-twl*  $S \equiv \text{image-mset } \text{raw-clause } (\text{learned-clss } (\text{rough-state-of-twl } S))$

**abbreviation** *backtrack-lvl-twl* **where**

*backtrack-lvl-twl*  $S \equiv \text{backtrack-lvl } (\text{rough-state-of-twl } S)$

**abbreviation** *conflicting-twl* **where**

*conflicting-twl*  $S \equiv \text{conflicting } (\text{rough-state-of-twl } S)$

**locale** *abstract-twl* =

**fixes**

*watch* :: ('v, nat, 'v clause) *twl-state*  $\Rightarrow$  'v clause  $\Rightarrow$  'v *twl-clause* **and**

*rewatch* :: ('v, nat, 'v literal multiset) *marked-lit*  $\Rightarrow$  ('v, nat, 'v clause) *twl-state*  $\Rightarrow$

'v *twl-clause*  $\Rightarrow$  'v *twl-clause* **and**

*linearize* :: 'v clauses  $\Rightarrow$  'v clause list **and**

*restart-learned* :: ('v, nat, 'v clause) *twl-state*  $\Rightarrow$  'v *twl-clause multiset*

**assumes**

*clause-watch*: *no-dup*(*trail*  $S$ )  $\implies$  *raw-clause* (*watch*  $S$   $C$ ) =  $C$  **and**

*wf-watch*: *no-dup* (*trail*  $S$ )  $\implies$  *wf-twl-cl*s (*trail*  $S$ ) (*watch*  $S$   $C$ ) **and**

*clause-rewatch*: *raw-clause* (*rewatch*  $L$   $S$   $C'$ ) = *raw-clause*  $C'$  **and**

*wf-rewatch*: *wf-twl-cl*s (*trail*  $S$ )  $C' \implies$  *wf-twl-cl*s ( $L \# \text{trail } S$ ) (*rewatch*  $L$   $S$   $C'$ ) **and**

*linearize*: *mset* (*linearize*  $N$ ) =  $N$  **and**

*restart-learned*: *restart-learned*  $S \subseteq \# \text{learned-clss } S$

**begin**

**lemma** *linearize-mempty[simp]*: *linearize*  $\{\#\} = []$

**using** *linearize mset-zero-iff* **by** *blast*

**definition**

*cons-trail* :: ('v, nat, 'v clause) *marked-lit*  $\Rightarrow$  ('v, nat, 'v clause) *twl-state*  $\Rightarrow$   
( 'v, nat, 'v clause) *twl-state*

**where**

*cons-trail*  $L$   $S =$

*TWL-State* ( $L \# \text{trail } S$ ) (*image-mset* (*rewatch*  $L$   $S$ ) (*init-clss*  $S$ ))

(*image-mset* (*rewatch*  $L$   $S$ ) (*learned-clss*  $S$ )) (*backtrack-lvl*  $S$ ) (*conflicting*  $S$ )

**definition**

*add-init-cl*s :: 'v clause  $\Rightarrow$  ('v, nat, 'v clause) *twl-state*  $\Rightarrow$   
( 'v, nat, 'v clause) *twl-state*

**where**

*add-init-cl*s  $C$   $S =$

*TWL-State* (*trail*  $S$ ) ( $\{\# \text{watch } S \ C \# \} + \text{init-clss } S$ ) (*learned-clss*  $S$ ) (*backtrack-lvl*  $S$ )  
(*conflicting*  $S$ )

**definition**

*add-learned-cl*s :: 'v clause  $\Rightarrow$  ('v, nat, 'v clause) *twl-state*  $\Rightarrow$   
( 'v, nat, 'v clause) *twl-state*

**where**

*add-learned-cl*s  $C$   $S =$

*TWL-State* (*trail*  $S$ ) (*init-clss*  $S$ ) ( $\{\# \text{watch } S \ C \# \} + \text{learned-clss } S$ ) (*backtrack-lvl*  $S$ )  
(*conflicting*  $S$ )

**definition**

*remove-cl*s :: 'v clause  $\Rightarrow$  ('v, nat, 'v clause) *twl-state*  $\Rightarrow$  ('v, nat, 'v clause) *twl-state*

**where**

*remove-cls*  $C$   $S$  =  
 $TWL-State$  (*trail*  $S$ ) (*filter-mset* ( $\lambda D. \text{raw-clause } D \neq C$ ) (*init-clss*  $S$ ))  
(*filter-mset* ( $\lambda D. \text{raw-clause } D \neq C$ ) (*learned-clss*  $S$ )) (*backtrack-lvl*  $S$ )  
(*conflicting*  $S$ )

**definition** *init-state* :: '*v* clauses  $\Rightarrow$  ('*v*, nat, '*v* clause) twl-state **where**

*init-state*  $N$  = *fold* *add-init-cls* (*linearize*  $N$ ) ( $TWL-State$  [] {#} {#} 0 *C-True*)

**lemma** *unchanged-fold-add-init-cls*:

*trail* (*fold* *add-init-cls*  $Cs$  ( $TWL-State$   $M$   $N$   $U$   $k$   $C$ )) =  $M$   
*learned-clss* (*fold* *add-init-cls*  $Cs$  ( $TWL-State$   $M$   $N$   $U$   $k$   $C$ )) =  $U$   
*backtrack-lvl* (*fold* *add-init-cls*  $Cs$  ( $TWL-State$   $M$   $N$   $U$   $k$   $C$ )) =  $k$   
*conflicting* (*fold* *add-init-cls*  $Cs$  ( $TWL-State$   $M$   $N$   $U$   $k$   $C$ )) =  $C$   
**by** (*induct*  $Cs$  *arbitrary*:  $N$ ) (*auto simp*: *add-init-cls-def*)

**lemma** *unchanged-init-state*[*simp*]:

*trail* (*init-state*  $N$ ) = []  
*learned-clss* (*init-state*  $N$ ) = {#}  
*backtrack-lvl* (*init-state*  $N$ ) = 0  
*conflicting* (*init-state*  $N$ ) = *C-True*  
**unfolding** *init-state-def* **by** (*rule* *unchanged-fold-add-init-cls*)+

**lemma** *clauses-init-fold-add-init*:

*no-dup*  $M \Rightarrow$   
*image-mset* *raw-clause* (*init-clss* (*fold* *add-init-cls*  $Cs$  ( $TWL-State$   $M$   $N$   $U$   $k$   $C$ ))) =  
*mset*  $Cs$  + *image-mset* *raw-clause*  $N$   
**by** (*induct*  $Cs$  *arbitrary*:  $N$ ) (*auto simp*: *add.assoc* *add-init-cls-def* *clause-watch*)

**lemma** *init-clss-init-state*[*simp*]: *image-mset* *raw-clause* (*init-clss* (*init-state*  $N$ )) =  $N$

**unfolding** *init-state-def* **by** (*simp* *add*: *clauses-init-fold-add-init* *linearize*)

**definition** *update-backtrack-lvl* **where**

*update-backtrack-lvl*  $k$   $S$  =  
 $TWL-State$  (*trail*  $S$ ) (*init-clss*  $S$ ) (*learned-clss*  $S$ )  $k$  (*conflicting*  $S$ )

**definition** *update-conflicting* **where**

*update-conflicting*  $C$   $S$  =  $TWL-State$  (*trail*  $S$ ) (*init-clss*  $S$ ) (*learned-clss*  $S$ ) (*backtrack-lvl*  $S$ )  $C$

**definition** *tl-trail* **where**

*tl-trail*  $S$  =  
 $TWL-State$  (*tl* (*trail*  $S$ )) (*init-clss*  $S$ ) (*learned-clss*  $S$ ) (*backtrack-lvl*  $S$ ) (*conflicting*  $S$ )

**definition** *restart'* **where**

*restart'*  $S$  =  $TWL-State$  [] (*init-clss*  $S$ ) (*restart-learned*  $S$ ) 0 *C-True*

**sublocale** *state<sub>W</sub>* *trail* *raw-init-clss* *raw-learned-clss* *backtrack-lvl* *conflicting*

*cons-trail* *tl-trail* *add-init-cls* *add-learned-cls* *remove-cls* *update-backtrack-lvl*  
*update-conflicting* *init-state* *restart'*

**apply** *unfold-locales*

**apply** (*simp-all* *add*: *add-init-cls-def* *add-learned-cls-def* *clause-rewatch* *clause-watch*  
*cons-trail-def* *remove-cls-def* *restart'-def* *tl-trail-def* *update-backtrack-lvl-def*  
*update-conflicting-def*)

**apply** (*rule* *image-mset-subseteq-mono*[*OF* *restart-learned*])

**done**

**sublocale**  $cdcl_W$ -ops trail raw-init-clss raw-learned-clsss backtrack-lvl conflicting  
 cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl  
 update-conflicting init-state restart'  
**by** unfold-locales

**interpretation**  $cdcl_{NOT}$ :  $cdcl_{NOT}$ -merge-bj-learn-ops convert-trail-from- $W$  o trail clauses  
 $\lambda L S. cons-trail (convert-marked-lit-from-NOT L) S$   
 $\lambda S. tl-trail S$   
 $\lambda C S. add-learned-cls C S$   
 $\lambda C S. remove-cls C S$   
 $\lambda L S. lit-of L \in fst \text{ 'candidates-propagate } S$   
 $\lambda S. conflicting S = C-True$   
 $\lambda C L S. C + \{\#L\# \} \in candidates-conflict S \wedge distinct-mset (C + \{\#L\# \}) \wedge \neg tautology (C + \{\#L\# \})$   
**by** unfold-locales

**end**

Lifting to the abstract state.

**context** abstract-twl  
**begin**

**declare** state-simp[simp del]

**abbreviation** cons-trail-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:  
 $undefined-lit (trail S) (lit-of L) \implies wf-twl-state S \implies wf-twl-state (cons-trail L S)$   
**unfolding** wf-twl-state-def **by** (auto simp: cons-trail-def wf-rewatch defined-lit-map)

**lemma** rough-state-of-twl-cons-trail:  
 $undefined-lit (trail-twl S) (lit-of L) \implies$   
 $rough-state-of-twl (cons-trail-twl L S) = cons-trail L (rough-state-of-twl S)$   
**using** rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-cons-trail **by** blast

**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:  $wf-twl-state S \implies wf-twl-state (add-init-cls L S)$   
**unfolding** wf-twl-state-def **by** (auto simp: wf-watch add-init-cls-def split: split-if-asm)

**lemma** rough-state-of-twl-add-init-cls:  
 $rough-state-of-twl (add-init-cls-twl L S) = add-init-cls L (rough-state-of-twl S)$   
**using** rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-init-cls **by** blast

**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:  $wf-twl-state S \implies wf-twl-state (add-learned-cls L S)$   
**unfolding** wf-twl-state-def **by** (auto simp: wf-watch add-learned-cls-def split: split-if-asm)

**lemma** rough-state-of-twl-add-learned-cls:  
 $rough-state-of-twl (add-learned-cls-twl L S) = add-learned-cls L (rough-state-of-twl S)$   
**using** rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-learned-cls **by** blast

**abbreviation** *remove-cls-twl* **where**

*remove-cls-twl C S*  $\equiv$  *twl-of-rough-state* (*remove-cls C* (*rough-state-of-twl S*))

**lemma** *wf-twl-remove-cls*: *wf-twl-state S*  $\implies$  *wf-twl-state* (*remove-cls L S*)

**unfolding** *wf-twl-state-def* **by** (*auto simp*: *wf-watch remove-cls-def split*: *split-if-asm*)

**lemma** *rough-state-of-twl-remove-cls*:

*rough-state-of-twl* (*remove-cls-twl L S*) = *remove-cls L* (*rough-state-of-twl S*)

**using** *rough-state-of-twl twl-of-rough-state-inverse wf-twl-remove-cls* **by** *blast*

**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-cls N S*)

**using** *assms apply* (*induction N arbitrary*: *S*)

**apply** (*auto simp*: *wf-twl-state-def*) $\square$

**by** (*simp add*: *wf-twl-add-init-cls*)

**lemma** *wf-twl-state-epsilon-state*[*simp*]:

*wf-twl-state* (*TWL-State*  $\square$   $\{\#\}$   $\{\#\}$  *O C-True*)

**by** (*auto simp*: *wf-twl-state-def*)

**lemma** *wf-twl-init-state*: *wf-twl-state* (*init-state N*)

**unfolding** *init-state-def* **by** (*auto intro!*: *wf-twl-state-wf-twl-state-fold-add-init-cls*)

**lemma** *rough-state-of-twl-init-state*:

*rough-state-of-twl* (*init-state-twl N*) = *init-state N*

**by** (*simp add*: *twl-of-rough-state-inverse wf-twl-init-state*)

**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*)

**by** (*simp add*: *twl-of-rough-state-inverse wf-twl-init-state wf-twl-cls-wf-twl-cls-tl tl-trail-def wf-twl-state-def distinct-tl map-tl*)

**lemma** *rough-state-of-twl-tl-trail*:

*rough-state-of-twl* (*tl-trail-twl S*) = *tl-trail* (*rough-state-of-twl S*)

**using** *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-tl-trail* **by** *blast*

**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*)

**unfolding** *wf-twl-state-def* **by** (*auto simp*: *update-backtrack-lvl-def*)

**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*)

**using** *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-backtrack-lvl* **by** *fast*

**abbreviation** *update-conflicting-twl* **where**  
*update-conflicting-twl*  $k$   $S \equiv \text{twl-of-rough-state } (\text{update-conflicting } k \text{ (rough-state-of-twl } S))$

**lemma** *wf-twl-state-update-conflicting*:  
*wf-twl-state*  $S \implies \text{wf-twl-state } (\text{update-conflicting } k \text{ } S)$   
**unfolding** *wf-twl-state-def* **by** (*auto simp: update-conflicting-def*)

**lemma** *rough-state-of-twl-update-conflicting*:  
*rough-state-of-twl* (*update-conflicting-twl*  $k$   $S$ ) = *update-conflicting*  $k$   
(*rough-state-of-twl*  $S$ )  
**using** *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-conflicting* **by** *fast*

**abbreviation** *raw-clauses-twl* **where**  
*raw-clauses-twl*  $S \equiv \text{clauses } (\text{rough-state-of-twl } S)$

**abbreviation** *restart-twl* **where**  
*restart-twl*  $S \equiv \text{twl-of-rough-state } (\text{restart}' \text{ (rough-state-of-twl } S))$

**lemma** *wf-wf-restart'*: *wf-twl-state*  $S \implies \text{wf-twl-state } (\text{restart}' \text{ } S)$   
**unfolding** *restart'-def wf-twl-state-def* **apply** *standard*  
**apply** *clarify*  
**apply** (*rename-tac*  $x$ )  
**apply** (*subgoal-tac wf-twl-cl* (*trail*  $S$ )  $x$ )  
**apply** (*case-tac*  $x$ )  
**using** *restart-learned* **by** *fastforce+*

**lemma** *rough-state-of-twl-restart-twl*:  
*rough-state-of-twl* (*restart-twl*  $S$ ) = *restart'* (*rough-state-of-twl*  $S$ )  
**by** (*simp add: twl-of-rough-state-inverse wf-wf-restart'*)

**interpretation** *cdcl<sub>NOT</sub>-twl-NOT*: *dpll-state*  
*convert-trail-from-W*  $o$  *trail-twl raw-clauses-twl*  
 $\lambda L$   $S$ . *cons-trail-twl* (*convert-marked-lit-from-NOT*  $L$ )  $S$   
 $\lambda S$ . *tl-trail-twl*  $S$   
 $\lambda C$   $S$ . *add-learned-cl*-*twl*  $C$   $S$   
 $\lambda C$   $S$ . *remove-cl*-*twl*  $C$   $S$   
**apply** *unfold-locales*  
**apply** (*simp add: rough-state-of-twl-cons-trail*)  
**apply** (*metis comp-apply rough-state-of-twl-tl-trail tl-trail*)  
**apply** (*metis comp-def rough-state-of-twl-add-learned-cl* *trail-add-cl*<sub>NOT</sub>)  
**apply** (*metis comp-apply rough-state-of-twl-remove-cl* *trail-remove-cl*)  
**apply** (*simp add: rough-state-of-twl-cons-trail*)  
**apply** (*metis clauses-tl-trail rough-state-of-twl-tl-trail*)  
**apply** (*simp add: rough-state-of-twl-add-learned-cl*)  
**using** *clauses-remove-cl*<sub>NOT</sub> *rough-state-of-twl-remove-cl* **by** *presburger*

**interpretation** *cdcl<sub>NOT</sub>-twl*: *state<sub>W</sub>*  
*trail-twl*  
*init-clss-twl*  
*learned-clss-twl*  
*backtrack-lvl-twl*  
*conflicting-twl*  
*cons-trail-twl*  
*tl-trail-twl*

*add-init-cls-twl*  
*add-learned-cls-twl*  
*remove-cls-twl*  
*update-backtrack-lvl-twl*  
*update-conflicting-twl*  
*init-state-twl*  
*restart-twl*  
**apply** *unfold-locales*  
**by** (*simp-all add: rough-state-of-twl-cons-trail rough-state-of-twl-tl-trail*  
*rough-state-of-twl-add-init-cls rough-state-of-twl-add-learned-cls rough-state-of-twl-remove-cls*  
*rough-state-of-twl-update-backtrack-lvl rough-state-of-twl-update-conflicting*  
*rough-state-of-twl-init-state rough-state-of-twl-restart-twl learned-clss-restart-state*)

**interpretation** *cdcl<sub>NOT</sub>-twl: cdcl<sub>W</sub>-ops*

*trail-twl*  
*init-clss-twl*  
*learned-clss-twl*  
*backtrack-lvl-twl*  
*conflicting-twl*  
*cons-trail-twl*  
*tl-trail-twl*  
*add-init-cls-twl*  
*add-learned-cls-twl*  
*remove-cls-twl*  
*update-backtrack-lvl-twl*  
*update-conflicting-twl*  
*init-state-twl*  
*restart-twl*  
**by** *unfold-locales*

**abbreviation** *state-eq-twl* (**infix**  $\sim_{TWL}$  51) **where**

*state-eq-twl S S'  $\equiv$  state-eq (rough-state-of-twl S) (rough-state-of-twl S')*

**notation** *cdcl<sub>NOT</sub>-twl.state-eq*(**infix**  $\sim$  51)

**declare** *cdcl<sub>NOT</sub>-twl.state-simp*[*simp del*]

**definition** *propagate-twl* **where**

*propagate-twl S S'  $\longleftrightarrow$*   
*( $\exists L C. (L, C) \in \text{candidates-propagate-twl } S$*   
 *$\wedge S' \sim_{TWL} \text{cons-trail-twl (Propagated } L \text{ } C) S$*   
 *$\wedge \text{conflicting-twl } S = C\text{-True}$ )*

**lemma** *propagate-twl-iff-propagate:*

**assumes** *inv: cdcl<sub>W</sub>-all-struct-inv (rough-state-of-twl S)*

**shows** *cdcl<sub>NOT</sub>-twl.propagate S T  $\longleftrightarrow$  propagate-twl S T* (**is**  $?P \longleftrightarrow ?T$ )

**proof**

**assume**  $?P$

**then obtain** *C L* **where**

*conflicting (rough-state-of-twl S) = C-True* **and**

*CL-Clauses:  $C + \{\#L\# \} \in \# \text{cdcl}_{NOT}\text{-twl.clauses } S$*  **and**

*tr-CNot: trail-twl S  $\models_{as} C\text{Not } C$*  **and**

*undef-lot: undefined-lit (trail-twl S) L* **and**

*T  $\sim \text{cons-trail-twl (Propagated } L \text{ } (C + \{\#L\# \})) S$*

**unfolding** *cdcl<sub>NOT</sub>-twl.propagate.simps* **by** *auto*



```

have distinct-mset ( $C + \{\#L\# \}$ )
  using inv CL-Clauses unfolding cdclW-all-struct-inv-def distinct-cdclW-state-def
  cdclNOT-twl.clauses-def distinct-mset-set-def
  by (metis (no-types, lifting) add-gr-0 mem-set-mset-iff plus-multiset.rep-eq)
then have C-L-L: mset-set (set-mset ( $C + \{\#L\# \}$ ) -  $\{L\}$ ) =  $C$ 
  by (metis Un-insert-right add-diff-cancel-left' add-diff-cancel-right'
    distinct-mset-set-mset-ident finite-set-mset insert-absorb2 mset-set.insert-remove
    set-mset-single set-mset-union)
have ( $L, C + \{\#L\# \}$ ) ∈ candidates-propagate-twl  $S$ 
  apply (rule wf-candidates-propagate-complete)
    using rough-state-of-twl apply auto[]
    using CL-Clauses cdclNOT-twl.clauses-def apply auto[]
  apply simp
  using C-L-L tr-CNot apply simp
  using undef-lot apply blast
done
show ?T unfolding propagate-twl-def
  apply (rule exI[of -  $L$ ], rule exI[of -  $C + \{\#L\# \}$ ])
  apply (auto simp:  $\langle (L, C + \{\#L\# \}) \in \text{candidates-propagate-twl } S \rangle$ 
     $\langle \text{conflicting (rough-state-of-twl } S) = C\text{-True} \rangle$ )
  using  $\langle T \sim \text{cons-trail-twl (Propagated } L (C + \{\#L\# \})) S \rangle$  cdclNOT-twl.state-eq-backtrack-lvl
  cdclNOT-twl.state-eq-conflicting cdclNOT-twl.state-eq-init-clss
  cdclNOT-twl.state-eq-learned-clss cdclNOT-twl.state-eq-trail state-eq-def by blast
next
assume ?T
then obtain  $L C$  where
  LC: ( $L, C$ ) ∈ candidates-propagate-twl  $S$  and
  T:  $T \sim \text{TWL cons-trail-twl (Propagated } L C) S$  and
  confl: conflicting (rough-state-of-twl  $S$ ) =  $C\text{-True}$ 
  unfolding propagate-twl-def by auto
have [simp]:  $C - \{\#L\# \} + \{\#L\# \} = C$ 
  using LC unfolding candidates-propagate-def
  by clarify (metis add commute add-diff-cancel-right' count-diff insert-DiffM
    multi-member-last not-gr0 zero-diff)
have  $C \in \# \text{ raw-clauses-twl } S$ 
  using LC unfolding candidates-propagate-def clauses-def by auto
then have distinct-mset  $C$ 
  using inv unfolding cdclW-all-struct-inv-def distinct-cdclW-state-def
  cdclNOT-twl.clauses-def distinct-mset-set-def clauses-def by auto
then have C-L-L: mset-set (set-mset  $C - \{L\}$ ) =  $C - \{\#L\# \}$ 
  by (metis  $\langle C - \{\#L\# \} + \{\#L\# \} = C \rangle$  add-left-imp-eq diff-single-trivial
    distinct-mset-set-mset-ident finite-set-mset mem-set-mset-iff mset-set.remove
    multi-self-add-other-not-self union-commute)

show ?P
  apply (rule cdclNOT-twl.propagate.intros[of - trail-twl  $S$  init-clss-twl  $S$ 
    learned-clss-twl  $S$  backtrack-lvl-twl  $S$   $C - \{\#L\# \}$   $L$ ])
    using confl apply auto[]
    using LC unfolding candidates-propagate-def apply (auto simp: cdclNOT-twl.clauses-def)[]
    using wf-candidates-propagate-sound[OF - LC] rough-state-of-twl apply (simp add: C-L-L)
    using wf-candidates-propagate-sound[OF - LC] rough-state-of-twl apply simp
  using T unfolding cdclNOT-twl.state-eq-def state-eq-def by auto
qed

```

definition conflict-twl where

```

conflict-tw1 S S'  $\longleftrightarrow$ 
  ( $\exists C. C \in \text{candidates-conflict-tw1 } S$ 
    $\wedge S' \sim \text{TWL update-conflicting-tw1 } (C\text{-Clause } C) S$ 
    $\wedge \text{conflicting-tw1 } S = C\text{-True}$ )

lemma conflict-tw1-iff-conflict:
  assumes inv: cdcl-all-struct-inv (rough-state-of-tw1 S)
  shows cdclNOT-tw1.conflict S T  $\longleftrightarrow$  conflict-tw1 S T (is ?C  $\longleftrightarrow$  ?T)
proof
  assume ?C
  then obtain M N U k C where
    S: state (rough-state-of-tw1 S) = (M, N, U, k, C-True) and
    C: C  $\in \#$  cdclNOT-tw1.clauses S and
    M-C: M  $\models_{as}$  CNot C and
    T: T  $\sim$  update-conflicting-tw1 (C-Clause C) S
  by auto
  have C  $\in$  candidates-conflict-tw1 S
  apply (rule wf-candidates-conflict-complete)
  apply simp
  using C apply (auto simp: cdclNOT-tw1.clauses-def)[]
  using M-C S by auto
  moreover have T  $\sim$  TWL tw1-of-rough-state (update-conflicting (C-Clause C) (rough-state-of-tw1 S))
  using T unfolding state-eq-def cdclNOT-tw1.state-eq-def by auto
  ultimately show ?T
  using S unfolding conflict-tw1-def by auto
next
  assume ?T
  then obtain C where
    C: C  $\in$  candidates-conflict-tw1 S and
    T: T  $\sim$  TWL update-conflicting-tw1 (C-Clause C) S and
    confl: conflicting-tw1 S = C-True
  unfolding conflict-tw1-def by auto
  have C  $\in \#$  cdclNOT-tw1.clauses S
  using C unfolding candidates-conflict-def cdclNOT-tw1.clauses-def by auto
  moreover have trail-tw1 S  $\models_{as}$  CNot C
  using wf-candidates-conflict-sound[OF - C] by auto
  ultimately show ?C apply -
  apply (rule cdclNOT-tw1.conflict.conflict-rule[of - - - - C])
  using confl T unfolding state-eq-def cdclNOT-tw1.state-eq-def by auto
qed

end

definition pull :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  pull p xs = filter p xs @ filter (Not  $\circ$  p) xs

lemma set-pull[simp]: set (pull p xs) = set xs
  unfolding pull-def by auto

lemma mset-pull[simp]: mset (pull p xs) = mset xs
  by (simp add: pull-def mset-filter-compl)

lemma mset-take-pull-sorted-list-of-set-subseteq:
  mset (take n (pull p (sorted-list-of-set (set-mset A))))  $\subseteq \#$  A
  by (metis mset-pull mset-set-set-mset-subseteq mset-sorted-list-of-set mset-take-subseteq)

```

*subset-mset.dual-order.trans*)

**definition** *watch-nat* :: (nat, nat, nat clause) twl-state  $\Rightarrow$  nat clause  $\Rightarrow$  nat twl-clause **where**  
*watch-nat* *S* *C* =

(let  
   *C'* = remdups (sorted-list-of-set (set-mset *C*));  
   negation-not-assigned = filter ( $\lambda L. -L \notin \text{ lits-of } (\text{trail } S)$ ) *C'*;  
   negation-assigned-sorted-by-trail = filter ( $\lambda L. L \in \# C$ ) (map ( $\lambda L. -\text{lit-of } L$ ) (trail *S*));  
   *W* = take 2 (negation-not-assigned @ negation-assigned-sorted-by-trail);  
   *UW* = sorted-list-of-multiset (*C* - mset *W*)  
 in TWL-Clause (mset *W*) (mset *UW*))

**definition**

*rewatch-nat* ::  
 (nat, nat, nat literal multiset) marked-lit  $\Rightarrow$  (nat, nat, nat clause) twl-state  $\Rightarrow$  nat twl-clause  $\Rightarrow$  nat twl-clause

**where**

*rewatch-nat* *L* *S* *C* =  
 (if - lit-of *L*  $\in \#$  watched *C* then  
   case filter ( $\lambda L'. L' \notin \#$  watched *C*  $\wedge - L' \notin \text{ lits-of } (L \# \text{ trail } S)$ )  
     (sorted-list-of-multiset (unwatched *C*)) of  
     []  $\Rightarrow$  *C*  
     | *L'* # -  $\Rightarrow$   
       TWL-Clause (watched *C* - {#- lit-of *L* #} + {#*L'* #}) (unwatched *C* - {#*L'* #} + {#- lit-of  
*L* #})  
   else  
     *C*)

**thm** *rev-cases*

**lemma** *list-cases2*:

**fixes** *l* :: 'a list

**assumes**

*l* = []  $\Rightarrow$  *P* **and**

$\bigwedge x. l = [x] \Rightarrow$  *P* **and**

$\bigwedge x y. l = [x, y] \Rightarrow$  *P* **and**

$\bigwedge x y xs. l = x \# y \# xs \Rightarrow$  *P*

**shows** *P*

**by** (metis assms(1) assms(2) assms(4) list.collapse)

**lemma** *XXX*:

**assumes** [*L*  $\leftarrow$  *P* . *L*  $\in \#$  *C*] = *l*

**shows**  $\forall x \in \text{set } l. x \in \text{set } P \wedge x \in \# C$

**using** assms **by** auto

**lemma** *XXX'*:

**assumes** [*L*  $\leftarrow$  *P* . *Q* *L*] = *l*

**shows**  $\forall x \in \text{set } l. x \in \text{set } P \wedge Q x$

**using** assms **by** auto

**lemma** *no-dup-filter-diff*:

**assumes** *n-d*: no-dup *M* **and** *H*: [*L*  $\leftarrow$  map ( $\lambda L. - \text{lit-of } L$ ) *M*. *L*  $\in \# C$ ] = *l*

**shows** distinct *l*

**unfolding** *H*[symmetric]

**apply** (rule distinct-filter)

**using** *n-d* **by** (induction *M*) auto

```

lemma mset-intersection-inclusion:  $A + (B - A) = B \longleftrightarrow A \subseteq\# B$ 
  apply (rule iffI)
  apply (metis mset-le-add-left)
  by (auto simp: ac-simps multiset-eq-iff subseteq-mset-def)

lemma clause-watch-nat: no-dup (trail S)  $\implies$  raw-clause (watch-nat S C) = C
  apply (simp add: watch-nat-def Let-def
    mset-intersection-inclusion)
  apply (cases [L $\leftarrow$ remdups (sorted-list-of-set (set-mset C)). - L  $\notin$  lits-of (trail S)] rule: list-cases2;
    cases [L $\leftarrow$ map ( $\lambda L$ . - lit-of L) (trail S) . L  $\in\#$  C] rule: list-cases2)
  apply (auto dest!: XXX' simp: ac-simps multiset-eq-iff)[]

apply (auto dest: XXX' dest: no-dup-filter-diff simp: ac-simps multiset-eq-iff)[1]
apply simp
apply (auto dest: XXX' dest: no-dup-filter-diff simp: ac-simps multiset-eq-iff)[]
sorry

lemma distinct-pull[simp]: distinct (pull p xs) = distinct xs
  unfolding pull-def by (induct xs) auto

lemma falsified-watched-imp-unwatched-falsified:
  assumes
    watched:  $L \in \text{set } (\text{take } n \text{ (pull (Not } \circ \text{ fls) (sorted-list-of-set (set-mset C))))}$  and
    falsified:  $\text{fls } L$  and
    not-watched:  $L' \notin \text{set } (\text{take } n \text{ (pull (Not } \circ \text{ fls) (sorted-list-of-set (set-mset C))))}$  and
    unwatched:  $L' \in\# C - \text{mset } (\text{take } n \text{ (pull (Not } \circ \text{ fls) (sorted-list-of-set (set-mset C))))}$ 
  shows  $\text{fls } L'$ 
proof -
  let ?Ls = sorted-list-of-set (set-mset C)
  let ?W = take n (pull (Not  $\circ$  fls) ?Ls)

  have  $n > \text{length } (\text{filter } (\text{Not } \circ \text{ fls}) ?Ls)$ 
  using watched falsified
  unfolding pull-def comp-def
  apply auto
  using in-set-takeD apply fastforce
  by (metis gr0I length-greater-0-conv length-pos-if-in-set take-0 zero-less-diff)
  then have  $\bigwedge L. L \in \text{set } ?Ls \implies \neg \text{fls } L \implies L \in \text{set } ?W$ 
  unfolding pull-def by auto
  then show ?thesis
  by (metis Multiset.diff-le-self finite-set-mset mem-set-mset-iff mset-leD not-watched
    sorted-list-of-set unwatched)
qed

lemma wf-watch-nat: no-dup (trail S)  $\implies$  wf-twl-cls (trail S) (watch-nat S C)
  apply (simp only: watch-nat-def Let-def partition-filter-conv case-prod-beta fst-conv snd-conv)
  unfolding wf-twl-cls.simps
  apply (intro conjI)
  apply clarsimp+
  using falsified-watched-imp-unwatched-falsified[unfolded comp-def]

sorry

```

```

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 -
  have  $x \in \text{set } ?comp$ 
    using assms by simp
  then have  $x \in \text{set } (\text{sorted-list-of-multiset } A)$ 
    by simp
  then show  $x \in \# A$ 
    by simp
qed

lemma clause-rewatch-nat:  $\text{raw-clause } (\text{rewatch-nat } L \ S \ C) = \text{raw-clause } C$ 
  apply (auto simp: rewatch-nat-def Let-def split: list.split)
  apply (subst subset-mset.add-diff-assoc2, simp)
  apply (subst subset-mset.add-diff-assoc2, simp)
  apply (subst subset-mset.add-diff-assoc2)
  apply (auto dest: filter-sorted-list-of-multiset-eqD)
  by (metis (no-types, lifting) add.assoc add-diff-cancel-right' filter-sorted-list-of-multiset-eqD
    insert-DiffM mset-leD mset-le-add-left)

lemma filter-sorted-list-of-multiset-Nil:
   $[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = [] \longleftrightarrow (\forall x \in \# M. \neg p \ x)$ 
  by auto (metis empty-iff filter-set list.set(1) mem-set-mset-iff member-filter
    set-sorted-list-of-multiset)

lemma filter-sorted-list-of-multiset-ConsD:
   $[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = x \ \# \ xs \implies p \ x$ 
  by (metis filter-set insert-iff list.set(2) member-filter)

lemma mset-minus-single-eq-mempty:
   $a - \{\#b\} = \{\#\} \longleftrightarrow a = \{\#b\} \vee a = \{\#\}$ 
  by (metis Multiset.diff-cancel add.right-neutral diff-single-eq-union
    diff-single-trivial zero-diff)

lemma wf-rewatch-nat':
  assumes wf:  $\text{wf-twl-cls } (\text{trail } S) \ C$ 
  shows  $\text{wf-twl-cls } (L \ \# \ \text{trail } S) \ (\text{rewatch-nat } L \ S \ C)$ 
using filter-sorted-list-of-multiset-Nil[simp]
proof (cases - lit-of L ∈ # watched C)
  case falsified: True

  let  $?unwatched\text{-nonfalsified} =$ 
     $[L' \leftarrow \text{sorted-list-of-multiset } (\text{unwatched } C). L' \notin \# \text{watched } C \wedge \neg L' \notin \text{lits-of } (L \ \# \ \text{trail } S)]$ 

  show  $?thesis$ 
proof (cases ?unwatched-nonfalsified)
  case Nil
  show  $?thesis$ 
    unfolding rewatch-nat-def
    using falsified Nil apply auto
    apply (case-tac C)
    apply auto
    using local.wf wf-twl-cls.simps apply blast

```

```

      using local.wf wf-twl-cls.simps apply blast
    sorry
  next
  case (Cons L' Ls)
  show ?thesis
    using wf
    unfolding rewatch-nat-def
    using falsified Cons

    sorry
  qed
next
case False
have wf-twl-cls (L # trail S) C
  using wf

  sorry
then show ?thesis
  unfolding rewatch-nat-def using False by simp
qed

interpretation abstract-twl watch-nat rewatch-nat sorted-list-of-multiset learned-clss
  apply unfold-locales
  apply (rule clause-watch-nat; simp)
  apply (rule wf-watch-nat; simp)
  apply (rule clause-rewatch-nat)
  apply (rule wf-rewatch-nat', simp)
  apply (rule mset-sorted-list-of-multiset)
  apply (rule subset-mset.order-refl)
oops

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin
sledgehammer-params[verbose]
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:
   $S \models_h C \longleftrightarrow \{\text{Pos } P \mid P. P \in S\} \cup \{\text{Neg } P \mid P. P \notin S\} \models C$ 
  unfolding Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def
  by auto

lemma herbrand-consistent-interp:
  consistent-interp ( $\{\text{Pos } P \mid P. P \in S\} \cup \{\text{Neg } P \mid P. P \notin S\}$ )
  unfolding consistent-interp-def by auto

lemma herbrand-total-over-set:

```

*total-over-set* ( $\{\text{Pos } P \mid P. P \in S\} \cup \{\text{Neg } P \mid P. P \notin S\}$ ) *T*  
**unfolding** *total-over-set-def* **by** *auto*

**lemma** *herbrand-total-over-m*:  
*total-over-m* ( $\{\text{Pos } P \mid P. P \in S\} \cup \{\text{Neg } P \mid P. P \notin S\}$ ) *T*  
**unfolding** *total-over-m-def* **by** (*auto simp add: herbrand-total-over-set*)

**lemma** *herbrand-interp-iff-partial-interp-clss*:  
 $S \models_{hs} C \longleftrightarrow \{\text{Pos } P \mid P. P \in S\} \cup \{\text{Neg } P \mid P. P \notin S\} \models_s C$   
**unfolding** *true-clss-def Ball-def herbrand-interp-iff-partial-interp-clss*  
*Partial-Clausal-Logic.true-clss-def* **by** *auto*

**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-selects-subseteq*:  $\bigwedge C. S \ C \leq \# C$  **and**  
*S-selects-neg-lits*:  $\bigwedge C \ L. L \in \# S \ C \implies \text{is-neg } L$

**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* =  
 $\{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max } (\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1$   
 $\wedge \neg (\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D) \models_h C \wedge S \ C = \{\#\}\}$   
**by** *auto*  
**termination by** (*relation*  $\{(D, C). D \# \subset \# C\}$ ) (*auto simp: wf-less-multiset*)

**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* =  $\{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max } (\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$   
*interp* *C*  $\models_h C \wedge S \ C = \{\#\}\}$   
**unfolding** *interp-def* **by** (*rule production.simps*)

**abbreviation** *productive* *A*  $\equiv$  (*production* *A*  $\neq \{\}$ )

**abbreviation** *produces* :: *'a clause*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool* **where**

*produces C A*  $\equiv$  *production C* = {A}

**lemma** *producesD*:

*produces C A*  $\implies C \in N \wedge C \neq \{\#\} \wedge \text{Pos } A = \text{Max } (\text{set-mset } C) \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$   
*interp C*  $\models_h C \wedge S \ C = \{\#\}$   
**unfolding** *production-unfold* **by** *auto*

**lemma** *produces C A*  $\implies \text{Pos } A \in \# \ C$

**by** (*simp add: Max-in-lits producesD*)

**lemma** *interp'-def-in-set*:

*interp C* =  $(\bigcup D \in \{D \in N. D \# \subseteq \# \ C\}. \text{production } D)$

**unfolding** *interp-def* **apply** *auto*

**unfolding** *production-unfold* **apply** *auto*

**done**

**lemma** *production-iff-produces*:

*produces D A*  $\longleftrightarrow A \in \text{production } D$

**unfolding** *production-unfold* **by** *auto*

**definition** *Interp* :: 'a *clause*  $\Rightarrow$  'a *interp* **where**

*Interp C* = *interp C*  $\cup$  *production C*

**lemma**

**assumes** *produces C P*

**shows** *Interp C*  $\models_h C$

**unfolding** *Interp-def* *assms* **using** *producesD[OF assms]*

**by** (*metis Max-in-lits Un-insert-right insertI1 pos-literal-in-imp-true-cls*)

**definition** *INTERP* :: 'a *interp* **where**

*INTERP* =  $(\bigcup D \in N. \text{production } D)$

**lemma** *interp-subseteq-Interp[simp]*: *interp C*  $\subseteq$  *Interp C*

**unfolding** *Interp-def* **by** *simp*

**lemma** *Interp-as-UNION*: *Interp C* =  $(\bigcup D \in \{D. D \# \subseteq \# \ C\}. \text{production } D)$

**unfolding** *Interp-def* *interp-def* *le-multiset-def* **by** *fast*

**lemma** *productive-not-empty*: *productive C*  $\implies C \neq \{\#\}$

**unfolding** *production-unfold* **by** *auto*

**lemma** *productive-imp-produces-Max-literal*: *productive C*  $\implies \text{produces } C (\text{atm-of } (\text{Max } (\text{set-mset } C)))$

**unfolding** *production-unfold* **by** (*auto simp del: atm-of-Max-lit*)

**lemma** *productive-imp-produces-Max-atom*: *productive C*  $\implies \text{produces } C (\text{Max } (\text{atms-of } C))$

**unfolding** *atms-of-def* *Max-atm-of-set-mset-commute[OF productive-not-empty]*

**by** (*rule productive-imp-produces-Max-literal*)

**lemma** *produces-imp-Max-literal*: *produces C A*  $\implies A = \text{atm-of } (\text{Max } (\text{set-mset } C))$

**by** (*metis Max-singleton insert-not-empty productive-imp-produces-Max-literal*)

**lemma** *produces-imp-Max-atom*: *produces C A*  $\implies A = \text{Max } (\text{atms-of } C)$

**by** (*metis Max-singleton insert-not-empty productive-imp-produces-Max-atom*)



**lemma** *produces-imp-Pos-in-lits*:  $\text{produces } C \ A \implies \text{Pos } A \in \# \ C$   
**by** (*auto intro: Max-in-lits dest!: producesD*)

**lemma** *productive-in-N*:  $\text{productive } C \implies C \in N$   
**unfolding** *production-unfold* **by** *auto*

**lemma** *produces-imp-atms-leq*:  $\text{produces } C \ A \implies B \in \text{atms-of } C \implies B \leq A$   
**by** (*metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom singleton-inject*)

**lemma** *produces-imp-neg-notin-lits*:  $\text{produces } C \ A \implies \neg \text{Neg } A \in \# \ C$   
**by** (*auto intro!: pos-Max-imp-neg-notin dest: producesD simp del: not-gr0*)

**lemma** *less-eq-imp-interp-subseteq-interp*:  $C \ \# \subseteq \# \ D \implies \text{interp } C \subseteq \text{interp } D$   
**unfolding** *interp-def* **by** *auto (metis multiset-order.order.strict-trans2)*

**lemma** *less-eq-imp-interp-subseteq-Interp*:  $C \ \# \subseteq \# \ D \implies \text{interp } C \subseteq \text{Interp } D$   
**unfolding** *Interp-def* **using** *less-eq-imp-interp-subseteq-interp* **by** *blast*

**lemma** *less-imp-production-subseteq-interp*:  $C \ \# \subset \# \ D \implies \text{production } C \subseteq \text{interp } D$   
**unfolding** *interp-def* **by** *fast*

**lemma** *less-eq-imp-production-subseteq-Interp*:  $C \ \# \subseteq \# \ D \implies \text{production } C \subseteq \text{Interp } D$   
**unfolding** *Interp-def* **using** *less-imp-production-subseteq-interp*  
**by** (*metis multiset-order.le-imp-less-or-eq le-supI1 sup-ge2*)

**lemma** *less-imp-Interp-subseteq-interp*:  $C \ \# \subset \# \ D \implies \text{Interp } C \subseteq \text{interp } D$   
**unfolding** *Interp-def*  
**by** (*auto simp: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)

**lemma** *less-eq-imp-Interp-subseteq-Interp*:  $C \ \# \subseteq \# \ D \implies \text{Interp } C \subseteq \text{Interp } D$   
**using** *less-imp-Interp-subseteq-interp*  
**unfolding** *Interp-def* **by** (*metis multiset-order.le-imp-less-or-eq le-supI2 subset-refl sup-commute*)

**lemma** *false-Interp-to-true-interp-imp-less-multiset*:  $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \ \# \subset \# \ D$   
**using** *less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

**lemma** *false-interp-to-true-interp-imp-less-multiset*:  $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \ \# \subset \# \ D$   
**using** *less-eq-imp-interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

**lemma** *false-Interp-to-true-Interp-imp-less-multiset*:  $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \ \# \subset \# \ D$   
**using** *less-eq-imp-Interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

**lemma** *false-interp-to-true-Interp-imp-le-multiset*:  $A \notin \text{interp } C \implies A \in \text{Interp } D \implies C \ \# \subseteq \# \ D$   
**using** *less-imp-Interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

**lemma** *interp-subseteq-INTERP*:  $\text{interp } C \subseteq \text{INTERP}$   
**unfolding** *interp-def INTERP-def* **by** (*auto simp: production-unfold*)

**lemma** *production-subseteq-INTERP*:  $\text{production } C \subseteq \text{INTERP}$   
**unfolding** *INTERP-def* **using** *production-unfold* **by** *blast*

**lemma** *Interp-subseteq-INTERP*:  $\text{Interp } C \subseteq \text{INTERP}$   
**unfolding** *Interp-def* **by** (*auto intro!: interp-subseteq-INTERP production-subseteq-INTERP*)

This lemma corresponds to theorem 2.7.6 page 66 of CW.

**lemma** *produces-imp-in-interp*:  
**assumes** *a-in-c*:  $Neg\ A \in\# C$  **and** *d*: *produces*  $D\ A$   
**shows**  $A \in\text{interp}\ C$   
**proof** –  
**from** *d* **have**  $Max\ (\text{set-mset}\ D) = Pos\ A$   
**using** *production-unfold* **by** *blast*  
**hence**  $D \# \subset \# \{\#Neg\ A\# \}$   
**by** (*auto intro: Max-pos-neg-less-multiset*)  
**moreover have**  $\{\#Neg\ A\# \} \# \subseteq \# C$   
**by** (*rule less-eq-imp-le-multiset*) (*rule mset-le-single[OF a-in-c[unfolded mem-set-mset-iff]]*)  
**ultimately show** *?thesis*  
**using** *d* **by** (*blast dest: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)  
**qed**

**lemma** *neg-notin-Interp-not-produce*:  $Neg\ A \in\# C \implies A \notin\text{Interp}\ D \implies C \# \subseteq \# D \implies \neg\ \text{produces}\ D''\ A$   
**by** (*auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-Interp*)

**lemma** *in-production-imp-produces*:  $A \in\text{production}\ C \implies \text{produces}\ C\ A$   
**by** (*metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq'*)

**lemma** *not-produces-imp-notin-production*:  $\neg\ \text{produces}\ C\ A \implies A \notin\text{production}\ C$   
**by** (*metis in-production-imp-produces*)

**lemma** *not-produces-imp-notin-interp*:  $(\bigwedge D. \neg\ \text{produces}\ D\ A) \implies A \notin\text{interp}\ C$   
**unfolding** *interp-def* **by** (*fast intro!: in-production-imp-produces*)

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** (*cases*  $\exists A. Pos\ A \in\# C \wedge A \in\text{Interp}\ D$ )  
**case** *True*  
**then obtain** *A* **where** *a-in-c*:  $Pos\ A \in\# C$  **and** *a-at-d*:  $A \in\text{Interp}\ D$   
**by** *blast*  
**from** *a-at-d* **have**  $A \in\text{interp}\ D'$   
**using** *d-lt-d'* *less-imp-Interp-subseteq-interp* **by** *blast*  
**thus** *?thesis*  
**using** *subs a-in-c* **by** (*blast dest: contra-subsetD*)  
**next**  
**case** *False*  
**then obtain** *A* **where** *a-in-c*:  $Neg\ A \in\# C$  **and**  $A \notin\text{Interp}\ D$   
**using** *c-at-d* *unfolding true-cls-def* **by** *blast*  
**hence**  $\bigwedge D''. \neg\ \text{produces}\ D''\ A$   
**using** *c-le-d* *neg-notin-Interp-not-produce* **by** *simp*  
**thus** *?thesis*  
**using** *a-in-c* *subs* *not-produces-imp-notin-production* **by** *auto*  
**qed**

**lemma** *true-Interp-imp-interp*:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp}\ D \models_h C \implies \text{interp}\ D' \models_h C$

```

using interp-def true-Interp-imp-general by simp

lemma true-Interp-imp-Interp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{Interp } D' \models_h C$ 
using Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general by simp

lemma true-Interp-imp-INTERP:  $C \# \subseteq \# D \implies \text{Interp } D \models_h C \implies \text{INTERP} \models_h C$ 
using INTERP-def interp-subseteq-INTERP
  true-Interp-imp-general[OF - less-multiset-right-total]
by simp

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 (cases  $\exists A. \text{Pos } A \in \# C \wedge A \in \text{interp } D$ )
case True
then obtain A where a-in-c:  $\text{Pos } A \in \# C$  and a-at-d:  $A \in \text{interp } D$ 
by blast
from a-at-d have  $A \in \text{interp } D'$ 
using d-lt-d' less-eq-imp-interp-subseteq-interp[OF multiset-order.less-imp-le] by blast
thus ?thesis
using subs a-in-c by (blast dest: contra-subsetD)
next
case False
then obtain A where a-in-c:  $\text{Neg } A \in \# C$  and  $A \notin \text{interp } D$ 
using c-at-d unfolding true-cls-def by blast
hence  $\bigwedge D''. \neg \text{produces } D'' A$ 
using c-le-d by (auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-interp)
thus ?thesis
using a-in-c subs not-produces-imp-notin-production by auto
qed

```

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

```

lemma true-interp-imp-interp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{interp } D' \models_h C$ 
using interp-def true-interp-imp-general by simp

```

```

lemma true-interp-imp-Interp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{Interp } D' \models_h C$ 
using Interp-as-UNION interp-subseteq-Interp[of D'] true-interp-imp-general by simp

```

```

lemma true-interp-imp-INTERP:  $C \# \subseteq \# D \implies \text{interp } D \models_h C \implies \text{INTERP} \models_h C$ 
using INTERP-def interp-subseteq-INTERP
  true-interp-imp-general[OF - less-multiset-right-total]
by simp

```

```

lemma productive-imp-false-interp:  $\text{productive } C \implies \neg \text{interp } C \models_h C$ 
unfolding production-unfold by auto

```

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

```

lemma cls-gt-double-pos-no-production:
assumes D:  $\{\# \text{Pos } P, \text{Pos } P\} \# \subset \# C$ 
shows  $\neg \text{produces } C P$ 
proof –

```

```

let ?D = {#Pos P, Pos P#}
note D' = D[unfolded less-multisetHO]
consider
  (P) count C (Pos P) ≥ 2
| (Q) Q where Q > Pos P and Q ∈# C
  using HOL.spec[OF HOL.conjunct2[OF D'], of Pos P] by auto
thus ?thesis
proof cases
  case Q
  have Q ∈ set-mset C
  using Q(2) by (auto split: split-if-asm)
  then have Max (set-mset C) > Pos P
  using Q(1) Max-gr-iff by blast
  thus ?thesis
  unfolding production-unfold by auto
next
  case P
  thus ?thesis
  unfolding production-unfold by auto
qed
qed

```

This lemma corresponds to theorem 2.7.6 page 66 of CW.

```

lemma
  assumes D: C + {#Neg P#} #⊂# D
  shows production D ≠ {P}
proof -
  note D' = D[unfolded less-multisetHO]
  consider
    (P) Neg P ∈# D
  | (Q) Q where Q > Neg P and count D Q > count (C + {#Neg P#}) Q
    using HOL.spec[OF HOL.conjunct2[OF D'], of Neg P] by fastforce
  thus ?thesis
  proof cases
    case Q
    have Q ∈ set-mset D
    using Q(2) by (auto split: split-if-asm)
    then have Max (set-mset D) > Neg P
    using Q(1) Max-gr-iff by blast
    hence Max (set-mset D) > Pos P
    using less-trans[of Pos P Neg P Max (set-mset D)] by auto
    thus ?thesis
    unfolding production-unfold by auto
  next
    case P
    hence Max (set-mset D) > Pos P
    by (meson Max-ge finite-set-mset le-less-trans linorder-not-le mem-set-mset-iff
      pos-less-neg)
    thus ?thesis
    unfolding production-unfold by auto
  qed
qed

```

```

lemma in-interp-is-produced:
  assumes P ∈ INTERP

```

**shows**  $\exists D. D + \{\#Pos\ P\# \} \in N \wedge produces\ (D + \{\#Pos\ P\# \})\ P$   
**using** *assms* **unfolding** *INTERP-def UN-iff production-iff-produces Ball-def*  
**by** (*metis ground-resolution-with-selection.produces-imp-Pos-in-lits insert-DiffM2*  
*ground-resolution-with-selection-axioms not-produces-imp-notin-production*)

**end**  
**end**

**abbreviation**  $MMax\ M \equiv Max\ (set-mset\ M)$

## 20.1 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]*:  $M < N \longleftrightarrow M \# \subset \# N$   
**unfolding** *less-multiset-def* **by** *auto*

**lemma** *less-eq-multiset[iff]*:  $M \leq N \longleftrightarrow M \# \subseteq \# N$   
**unfolding** *less-eq-multiset-def* **by** *auto*

**lemma** *herbrand-true-clss-true-clss-clss-herbrand-true-clss*:

**assumes**  
 $AB: A \models_{hs} B$  **and**  
 $BC: B \models_p C$   
**shows**  $A \models_h C$

**proof** –

**let**  $?I = \{Pos\ P\ | P. P \in A\} \cup \{Neg\ P\ | P. P \notin A\}$   
**have**  $B: ?I \models_s B$  **using**  $AB$   
**by** (*auto simp add: herbrand-interp-iff-partial-interp-clss*)

**have**  $IH: \bigwedge I. total-over-set\ I\ (atms-of\ C) \Longrightarrow total-over-m\ I\ B \Longrightarrow consistent-interp\ I$   
 $\Longrightarrow I \models_s B \Longrightarrow I \models C$  **using**  $BC$   
**by** (*auto simp add: true-clss-clss-def*)

**show** *?thesis*

**unfolding** *herbrand-interp-iff-partial-interp-clss*  
**by** (*auto intro: IH[of ?I] simp add: herbrand-total-over-set herbrand-total-over-m*  
*herbrand-consistent-interp B*)

**qed**

**lemma** *abstract-red-subset-mset-abstract-red*:

**assumes**  
 $abstr: abstract-red\ C\ N$  **and**  
 $c-lt-d: C \subseteq \# D$   
**shows**  $abstract-red\ D\ N$

**proof** –

**have**  $\{D \in N. D \# \subset \# C\} \subseteq \{D' \in N. D' \# \subset \# D\}$

```

  using c-lt-d less-eq-imp-le-multiset by fastforce
thus ?thesis
  using abstr unfolding abstract-red-def clss-lt-def
  by (metis (no-types, lifting) c-lt-d subset-mset.diff-add true-clss-cls-mono-r'
      true-clss-cls-subset)
qed

```

**lemma** *true-clss-cls-extended*:

```

assumes
  A  $\models_p$  B and
  tot: total-over-m I (A) and
  cons: consistent-interp I and
  I-A: I  $\models_s$  A
shows I  $\models$  B
proof -
  let ?I = I  $\cup$  {Pos P | P. P  $\in$  atms-of B  $\wedge$  P  $\notin$  atms-of-s I}
  have consistent-interp ?I
  using cons unfolding consistent-interp-def atms-of-s-def atms-of-def
  apply (auto 1 5 simp add: image-iff)
  by (metis atm-of-uminus literal.sel(1))
moreover have total-over-m ?I (A  $\cup$  {B})
proof -
  obtain aa :: 'a set  $\Rightarrow$  'a literal set  $\Rightarrow$  'a where
    f2:  $\forall x0 x1. (\exists v2. v2 \in x0 \wedge \text{Pos } v2 \notin x1 \wedge \text{Neg } v2 \notin x1) \iff (aa \ x0 \ x1 \in x0 \wedge \text{Pos } (aa \ x0 \ x1) \notin x1 \wedge \text{Neg } (aa \ x0 \ x1) \notin x1)$ 
  by maura
  have  $\forall a. a \notin \text{atms-of-m } A \vee \text{Pos } a \in I \vee \text{Neg } a \in I$ 
  using tot by (simp add: total-over-m-def total-over-set-def)
  hence aa (atms-of-m A  $\cup$  atms-of-m {B}) (I  $\cup$  {Pos a | a. a  $\in$  atms-of B  $\wedge$  a  $\notin$  atms-of-s I})
   $\notin$  atms-of-m A  $\cup$  atms-of-m {B}  $\vee$  Pos (aa (atms-of-m A  $\cup$  atms-of-m {B})
    (I  $\cup$  {Pos a | a. a  $\in$  atms-of B  $\wedge$  a  $\notin$  atms-of-s I}))  $\in$  I
     $\cup$  {Pos a | a. a  $\in$  atms-of B  $\wedge$  a  $\notin$  atms-of-s I}
     $\vee$  Neg (aa (atms-of-m A  $\cup$  atms-of-m {B})
      (I  $\cup$  {Pos a | a. a  $\in$  atms-of B  $\wedge$  a  $\notin$  atms-of-s I}))  $\in$  I
       $\cup$  {Pos a | a. a  $\in$  atms-of B  $\wedge$  a  $\notin$  atms-of-s I}
  by auto
  hence total-over-set (I  $\cup$  {Pos a | a. a  $\in$  atms-of B  $\wedge$  a  $\notin$  atms-of-s I}) (atms-of-m A  $\cup$  atms-of-m {B})
  using f2 by (meson total-over-set-def)
  thus ?thesis
  by (simp add: total-over-m-def)
qed
moreover have ?I  $\models_s$  A
using I-A by auto
ultimately have ?I  $\models$  B
using  $\langle A \models_p B \rangle$  unfolding true-clss-cls-def by auto
thus ?thesis

```

**oops**

**lemma**

```

assumes
  CP:  $\neg$  clss-lt N ({#C#} + {#E#})  $\models_p$  {#C#} + {#Neg P#} and
  clss-lt N ({#C#} + {#E#})  $\models_p$  {#E#} + {#Pos P#}  $\vee$  clss-lt N ({#C#} + {#E#})  $\models_p$ 
  {#C#} + {#Neg P#}
shows clss-lt N ({#C#} + {#E#})  $\models_p$  {#E#} + {#Pos P#}

```

oops

```

locale ground-ordered-resolution-with-redundancy =
  ground-resolution-with-selection +
  fixes redundant :: 'a::wellorder clause  $\Rightarrow$  'a clauses  $\Rightarrow$  bool
  assumes
    redundant-iff-abstract: redundant A N  $\longleftrightarrow$  abstract-red A N
begin
definition saturated :: 'a clauses  $\Rightarrow$  bool where
  saturated N  $\longleftrightarrow$  ( $\forall$  A B C. A  $\in$  N  $\longrightarrow$  B  $\in$  N  $\longrightarrow$   $\neg$ redundant A N  $\longrightarrow$   $\neg$ redundant B N
     $\longrightarrow$  superposition-rules A B C  $\longrightarrow$  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
proof (rule ccontr)
  let  $?N_{\mathcal{I}} = \text{INTERP } N$ 
  assume  $\neg ?thesis$ 
  hence not-empty:  $\{E \in N. \neg ?N_{\mathcal{I}} \models_h E\} \neq \{\}$ 
    unfolding true-clss-def Ball-def by auto
  def D  $\equiv$  Min  $\{E \in N. \neg ?N_{\mathcal{I}} \models_h E\}$ 
  have [simp]: D  $\in$  N
    unfolding D-def
    by (metis (mono-tags, lifting) Min-in not-empty finite mem-Collect-eq rev-finite-subset subsetI)
  have not-d-interp:  $\neg ?N_{\mathcal{I}} \models_h D$ 
    unfolding D-def
    by (metis (mono-tags, lifting) Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI)
  have cls-not-D:  $\bigwedge E. E \in N \Longrightarrow E \neq D \Longrightarrow \neg ?N_{\mathcal{I}} \models_h E \Longrightarrow D \leq E$ 
    using finite D-def by (auto simp del: less-eq-multiset)
  obtain C L where D: D = C +  $\{\#L\#\}$  and LSD:  $L \in \# S D \vee (S D = \{\#\} \wedge \text{Max } (set-mset D) = L)$ 
  proof (cases  $S D = \{\#\}$ )
    case False
    then obtain L where  $L \in \# S D$ 
      using Max-in-lits by blast
    moreover
      hence  $L \in \# D$ 
        using S-selects-subseteq[of D] by auto
      hence  $D = (D - \{\#L\#\}) + \{\#L\#\}$ 
        by auto
      ultimately show ?thesis using that by blast
    next
    let  $?L = \text{MMax } D$ 
    case True
    moreover
      have  $?L \in \# D$ 
        by (metis (no-types, lifting) Max-in-lits  $\langle D \in N \rangle$  empty)
      hence  $D = (D - \{\#?L\#\}) + \{\#?L\#\}$ 
        by auto
      ultimately show ?thesis using that by blast
  qed

```

```

have red:  $\neg$ redundant  $D$   $N$ 
proof (rule ccontr)
  assume red[simplified]:  $\sim\sim$ redundant  $D$   $N$ 
  have  $\forall E < D. E \in N \longrightarrow ?N_{\mathcal{I}} \models_h E$ 
    using cls-not- $D$  not-le by fastforce
  hence  $?N_{\mathcal{I}} \models_{hs} \text{clss-lt } N D$ 
    unfolding clss-lt-def true-clss-def Ball-def by blast
  thus False
    using red not-d-interp unfolding abstract-red-def redundant-iff-abstract
    using herbrand-true-clss-true-clss-cls-herbrand-true-clss by fast
qed

consider
  (L)  $P$  where  $L = \text{Pos } P$  and  $S D = \{\#\}$  and  $\text{Max } (\text{set-mset } D) = \text{Pos } P$ 
| (Lneg)  $P$  where  $L = \text{Neg } P$ 
  using LSD  $S$ -selects-neg-lits[of  $D$   $L$ ] by (cases  $L$ ) auto
thus False
proof cases
  case L note  $P = \text{this}(1)$  and  $S = \text{this}(2)$  and  $\text{max} = \text{this}(3)$ 
  have count  $D$   $L > 1$ 
  proof (rule ccontr)
    assume  $\sim ?thesis$ 
    hence count: count  $D$   $L = 1$ 
    unfolding  $D$  by auto
    have  $\neg ?N_{\mathcal{I}} \models_h D$ 
    using not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms
    by blast
    hence produces  $N$   $D$   $P$ 
    using not-empty empty finite  $\langle D \in N \rangle$  count  $L$ 
    true-interp-imp-INTERP unfolding production-iff-produces unfolding production-unfold
    by (auto simp add: max not-empty)
    hence INTERP  $N \models_h D$ 
    unfolding  $D$ 
    by (metis pos-literal-in-imp-true-cls produces-imp-Pos-in-lits
    production-subseteq-INTERP singletonI subsetCE)
    thus False
    using not-d-interp by blast
  qed
  then obtain  $C'$  where  $C':D = C' + \{\#\text{Pos } P\} + \{\#\text{Pos } P\}$ 
    unfolding  $D$  by (metis  $P$  add.left-neutral add-less-cancel-right count-single count-union
    multi-member-split)
  have sup: superposition-rules  $D$   $D$  ( $D - \{\#L\}$ )
    unfolding  $C' L$  by (auto simp add: superposition-rules.simps)
  have  $C' + \{\#\text{Pos } P\} \# \subset \# C' + \{\#\text{Pos } P\} + \{\#\text{Pos } P\}$ 
    by auto
  moreover have  $\neg ?N_{\mathcal{I}} \models_h (D - \{\#L\})$ 
    using not-d-interp unfolding  $C' L$  by auto
  ultimately have  $C' + \{\#\text{Pos } P\} \notin N$ 
    by (metis (no-types, lifting)  $C' P$  add-diff-cancel-right' cls-not- $D$  less-multiset
    multi-self-add-other-not-self not-le)
  have  $D - \{\#L\} \# \subset \# D$ 
    unfolding  $C' L$  by auto
  have  $c'-p-p: C' + \{\#\text{Pos } P\} + \{\#\text{Pos } P\} - \{\#\text{Pos } P\} = C' + \{\#\text{Pos } P\}$ 
    by auto
  have redundant  $(C' + \{\#\text{Pos } P\})$   $N$ 

```



**using** *saturated red sup*  $\langle D \in N \rangle \langle C' + \{\#Pos P\# \} \notin N \rangle$  **unfolding** *saturated-def*  $C' L c'-p-p$   
**by** *blast*  
**moreover have**  $C' + \{\#Pos P\# \} \subseteq\# C' + \{\#Pos P\# \} + \{\#Pos P\# \}$   
**by** *auto*  
**ultimately show** *False*  
**using** *red* **unfolding**  $C'$  *redundant-iff-abstract* **by** (*blast dest:*  
*abstract-red-subset-mset-abstract-red*)  
**next**  
**case**  $Lneg$  **note**  $L = this(1)$   
**have**  $P \in ?N_{\mathcal{I}}$   
**using** *not-d-interp* **unfolding**  $D$  *true-cls-def*  $L$  **by** (*auto split: split-if-asm*)  
**then obtain**  $E$  **where**  
 $DPN: E + \{\#Pos P\# \} \in N$  **and**  
 $prod: production\ N\ (E + \{\#Pos P\# \}) = \{P\}$   
**using** *in-interp-is-produced* **by** *blast*  
**have**  $sup-EC: superposition-rules\ (E + \{\#Pos P\# \})\ (C + \{\#Neg P\# \})\ (E + C)$   
**using** *superposition-l* **by** *fast*  
**hence**  $superposition\ N\ (N \cup \{E+C\})$   
**using**  $DPN \langle D \in N \rangle$  **unfolding**  $D L$  **by** (*auto simp add: superposition.simps*)  
**have**  
 $PMax: Pos\ P = MMax\ (E + \{\#Pos P\# \})$  **and**  
 $count\ (E + \{\#Pos P\# \})\ (Pos\ P) \leq 1$  **and**  
 $S\ (E + \{\#Pos P\# \}) = \{\#\}$  **and**  
 $\neg interp\ N\ (E + \{\#Pos P\# \}) \models_h E + \{\#Pos P\# \}$   
**using**  $prod$  **unfolding** *production-unfold* **by** *auto*  
**have**  $Neg\ P \notin\# E$   
**using**  $prod$  *produces-imp-neg-notin-lits* **by** *force*  
**hence**  $\bigwedge y. y \in\# (E + \{\#Pos P\# \})$   
 $\implies count\ (E + \{\#Pos P\# \})\ (Neg\ P) < count\ (C + \{\#Neg P\# \})\ (Neg\ P)$   
**by** (*auto split: split-if-asm*)  
**moreover have**  $\bigwedge y. y \in\# (E + \{\#Pos P\# \}) \implies y < Neg\ P$   
**using**  $PMax$  **by** (*metis DPN Max-less-iff empty finite-set-mset mem-set-mset-iff pos-less-neg*  
*set-mset-eq-empty-iff*)  
**moreover have**  $E + \{\#Pos P\# \} \neq C + \{\#Neg P\# \}$   
**using**  $prod$  *produces-imp-neg-notin-lits* **by** *force*  
**ultimately have**  $E + \{\#Pos P\# \} \# \subset\# C + \{\#Neg P\# \}$   
**unfolding** *less-multiset<sub>HO</sub>* **by** (*metis add.left-neutral add-lessD1*)  
**have**  $ce-lt-d: C + E \# \subset\# D$   
**unfolding**  $D L$   
**by** (*metis (mono-tags, lifting) Max-pos-neg-less-multiset One-nat-def PMax count-single*  
*less-multiset-plus-right-nonempty mult-less-trans single-not-empty union-less-mono2*  
*zero-less-Suc*)  
**have**  $?N_{\mathcal{I}} \models_h E + \{\#Pos P\# \}$   
**using**  $\langle P \in ?N_{\mathcal{I}} \rangle$  **by** *blast*  
**have**  $?N_{\mathcal{I}} \models_h C+E \vee C+E \notin N$   
**using**  $ce-lt-d$  *cls-not-D* **unfolding**  $D-def$  **by** *fastforce*  
**have**  $Pos\ P \notin\# C+E$   
**using**  $D \langle P \in ground-resolution-with-selection.INTERP\ S\ N \rangle$   
 $\langle count\ (E + \{\#Pos P\# \})\ (Pos\ P) \leq 1 \rangle$  *multi-member-skip not-d-interp* **by** *auto*  
**hence**  $\bigwedge y. y \in\# C+E$   
 $\implies count\ (C+E)\ (Pos\ P) < count\ (E + \{\#Pos P\# \})\ (Pos\ P)$   
**by** (*auto split: split-if-asm*)  
  
**have**  $\neg redundant\ (C + E)\ N$   
**proof** (*rule ccontr*)

```

assume  $red'[simplified]: \neg ?thesis$ 
have  $abs: clss-lt\ N\ (C + E) \models_p C + E$ 
  using redundant-iff-abstract  $red'$  unfolding abstract-red-def by auto
have  $clss-lt\ N\ (C + E) \models_p E + \{\#Pos\ P\# \} \vee clss-lt\ N\ (C + E) \models_p C + \{\#Neg\ P\# \}$ 
proof clarify
  assume  $CP: \neg clss-lt\ N\ (C + E) \models_p C + \{\#Neg\ P\# \}$ 
  { fix  $I$ 
    assume
       $total-over-m\ I\ (clss-lt\ N\ (C + E) \cup \{E + \{\#Pos\ P\# \}\})$  and
       $consistent-interp\ I$  and
       $I \models_s clss-lt\ N\ (C + E)$ 
      hence  $I \models C + E$ 
      using abs sorry
      moreover have  $\neg I \models C + \{\#Neg\ P\# \}$ 
      using  $CP$  unfolding true-clss-clss-def
      sorry
      ultimately have  $I \models E + \{\#Pos\ P\# \}$  by auto
    }
  then show  $clss-lt\ N\ (C + E) \models_p E + \{\#Pos\ P\# \}$ 
    unfolding true-clss-clss-def by auto
  qed
moreover have  $clss-lt\ N\ (C + E) \subseteq clss-lt\ N\ (C + \{\#Neg\ P\# \})$ 
  using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
ultimately have  $redundant\ (C + \{\#Neg\ P\# \})\ N \vee clss-lt\ N\ (C + E) \models_p E + \{\#Pos\ P\# \}$ 
  unfolding redundant-iff-abstract abstract-red-def using true-clss-clss-subset by blast
show False sorry
qed
moreover have  $\neg redundant\ (E + \{\#Pos\ P\# \})\ N$ 
sorry
ultimately have  $CEN: C + E \in N$ 
  using  $\langle D \in N \rangle \langle E + \{\#Pos\ P\# \} \in N \rangle$  saturated sup-EC  $red$  unfolding saturated-def D L
  by (metis union-commute)
have  $CED: C + E \neq D$ 
  using  $D$  ce-lt-d by auto
have  $interp: \neg INTERP\ N \models_h C + E$ 
sorry
show False
  using cls-not-D[OF CEN CED interp] ce-lt-d unfolding INTERP-def less-eq-multiset-def by
auto
qed
qed

```

end

**lemma** *tautology-is-redundant:*

```

assumes tautology C
shows abstract-red C N
using assms unfolding abstract-red-def true-clss-clss-def tautology-def by auto

```

**lemma** *subsumed-is-redundant:*

```

assumes  $AB: A \subset\# B$ 
and  $AN: A \in N$ 
shows abstract-red B N

```

**proof** —

```

have  $A \in clss-lt\ N\ B$  using  $AN\ AB$  unfolding clss-lt-def

```

```

    by (auto dest: less-eq-imp-le-multiset simp add: multiset-order.dual-order.order-iff-strict)
  thus ?thesis
    using AB unfolding abstract-red-def true-clss-clss-def Partial-Clausal-Logic.true-clss-def
    by blast
qed

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
  using assms
proof (induction rule: redundant.induct)
  case (subsumption A B N)
  thus ?case
    using subsumed-is-redundant[of A N B] unfolding abstract-red-def clss-lt-def by auto
qed

lemma redundant-mono:
   $\text{redundant } A N \Longrightarrow A \subseteq\# B \Longrightarrow \text{redundant } B N$ 
  apply (induction rule: redundant.induct)
  by (meson subset-mset.less-le-trans subsumption)

locale truc=
  selection S for S :: nat clause  $\Rightarrow$  nat clause
begin

end

end
theory Weidenbach-Book
imports
  Prop-Normalisation

  Prop-Resolution

  Prop-Superposition

  CDCL-NOT DPLL-NOT DPLL-W-Implementation CDCL-W-Implementation CDCL-W-Incremental
  CDCL-WNOT CDCL-Two-Watched-Literals

begin

end

```