

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

February 11, 2016

Contents

| | | |
|----------|--|-----------|
| 1 | Transitions | 5 |
| 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 |
| 2 | Various Lemmas | 10 |
| 3 | More List | 12 |
| 3.1 | <i>upt</i> | 12 |
| 3.2 | Lexicographic ordering | 14 |
| 4 | Logics | 14 |
| 4.1 | Definition and abstraction | 14 |
| 4.2 | properties of the abstraction | 16 |
| 4.3 | Subformulas and properties | 18 |
| 4.4 | Positions | 21 |
| 5 | Semantics over the syntax | 24 |
| 6 | Rewrite systems and properties | 26 |
| 6.1 | Lifting of rewrite rules | 26 |
| 6.2 | Consistency preservation | 28 |
| 6.3 | Full Lifting | 29 |
| 7 | Transformation testing | 30 |
| 7.1 | Definition and first properties | 30 |
| 7.2 | Invariant conservation | 32 |
| 7.2.1 | Invariant while lifting of the rewriting relation | 33 |
| 7.2.2 | Invariant after all rewriting | 34 |
| 8 | Rewrite Rules | 36 |
| 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 |
| 9 | The full transformations | 64 |
| 9.1 | Abstract Property characterizing that only some connective are inside the others | 64 |
| 9.1.1 | Definition | 64 |
| 9.2 | Conjunctive Normal Form | 67 |
| 9.2.1 | Full CNF transformation | 67 |
| 9.3 | Disjunctive Normal Form | 68 |
| 9.3.1 | Full DNF transform | 68 |
| 10 | More aggressive simplifications: Removing true and false at the beginning | 68 |
| 10.1 | Transformation | 68 |
| 10.2 | More invariants | 70 |
| 10.3 | The new CNF and DNF transformation | 74 |
| 11 | Partial Clausal Logic | 75 |
| 11.1 | Clauses | 75 |
| 11.2 | Partial Interpretations | 75 |
| 11.2.1 | Consistency | 76 |
| 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 |
| 12 | Resolution | 100 |
| 12.1 | Simplification Rules | 100 |
| 12.2 | Unconstrained Resolution | 102 |
| 12.2.1 | Subsumption | 103 |
| 12.3 | Inference Rule | 103 |
| 12.4 | Lemma about the simplified state | 118 |
| 12.5 | Resolution and Invariants | 121 |
| 12.5.1 | Invariants | 121 |
| 12.5.2 | well-foundedness if the relation | 127 |
| 13 | Partial Clausal Logic | 142 |
| 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 | 158 |
| 14 | NOT's CDCL | 159 |
| 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 | 166 |
| 14.3 | DPLL with backjumping | 167 |
| 14.3.1 | Definition | 167 |
| 14.3.2 | Basic properties | 168 |
| 14.3.3 | Termination | 171 |
| 14.3.4 | Normal Forms | 176 |
| 14.4 | CDCL | 183 |
| 14.4.1 | Learn and Forget | 183 |
| 14.4.2 | Definition of CDCL | 184 |
| 14.5 | CDCL with invariant | 187 |
| 14.6 | Termination | 193 |
| 14.6.1 | Restricting learn and forget | 193 |
| 14.7 | CDCL with restarts | 204 |
| 14.7.1 | Definition | 204 |
| 14.7.2 | Increasing restarts | 205 |
| 14.8 | Merging backjump and learning | 212 |
| 14.8.1 | Instantiations | 223 |
| 15 | DPLL as an instance of NOT | 239 |
| 15.1 | DPLL with simple backtrack | 239 |
| 15.2 | Adding restarts | 243 |
| 16 | DPLL | 244 |
| 16.1 | Rules | 244 |
| 16.2 | Invariants | 244 |
| 16.3 | Termination | 253 |
| 16.4 | Final States | 255 |
| 16.5 | Link with NOT's DPLL | 256 |
| 16.5.1 | Level of literals and clauses | 258 |
| 16.5.2 | Properties about the levels | 262 |
| 17 | Weidenbach's CDCL | 265 |
| 17.1 | The State | 265 |
| 17.2 | Special Instantiation: using Triples as State | 271 |
| 17.3 | CDCL Rules | 271 |
| 17.4 | Invariants | 277 |
| 17.4.1 | Properties of the trail | 277 |
| 17.4.2 | Better-Suited Induction Principle | 281 |
| 17.4.3 | Compatibility with $op \sim$ | 285 |
| 17.4.4 | Conservation of some Properties | 287 |
| 17.4.5 | Learned Clause | 288 |
| 17.4.6 | No alien atom in the state | 289 |
| 17.4.7 | No duplicates all around | 292 |

| | | |
|---------------------------------------|--|------------|
| 17.4.8 | Conflicts and co | 293 |
| 17.4.9 | Putting all the invariants together | 301 |
| 17.4.10 | No tautology is learned | 304 |
| 17.5 | CDCL Strong Completeness | 305 |
| 17.6 | Higher level strategy | 306 |
| 17.6.1 | Definition | 306 |
| 17.6.2 | Invariants | 309 |
| 17.6.3 | Literal of highest level in conflicting clauses | 314 |
| 17.6.4 | Literal of highest level in marked literals | 318 |
| 17.6.5 | Strong completeness | 327 |
| 17.6.6 | No conflict with only variables of level less than backtrack level | 333 |
| 17.6.7 | Final States are Conclusive | 344 |
| 17.7 | Termination | 350 |
| 17.8 | No Relearning of a clause | 351 |
| 17.9 | Decrease of a measure | 366 |
| 18 | Simple Implementation of the DPLL and CDCL | 372 |
| 18.1 | Common Rules | 372 |
| 18.1.1 | Propagation | 372 |
| 18.1.2 | Unit propagation for all clauses | 374 |
| 18.1.3 | Decide | 375 |
| 18.2 | Simple Implementation of DPLL | 376 |
| 18.2.1 | Combining the propagate and decide: a DPLL step | 376 |
| 18.2.2 | Adding invariants | 378 |
| 18.2.3 | Code export | 385 |
| 18.3 | CDCL Implementation | 387 |
| 18.3.1 | Definition of the rules | 387 |
| 18.3.2 | Propagate | 389 |
| 18.3.3 | Code generation | 400 |
| 19 | Link between Weidenbach's and NOT's CDCL | 412 |
| 19.1 | Inclusion of the states | 412 |
| 19.2 | Additional Lemmas between NOT and W states | 417 |
| 19.3 | More lemmas conflict-propagate and backjumping | 417 |
| 19.3.1 | Termination | 417 |
| 19.3.2 | More backjumping | 418 |
| 19.4 | CDCL FW | 431 |
| 19.5 | FW with strategy | 441 |
| 19.5.1 | The intermediate step | 441 |
| 19.6 | Adding Restarts | 477 |
| 20 | Incremental SAT solving | 487 |
| 20.1 | We can now define the rules of the calculus | 535 |
| theory <i>Wellfounded-More</i> | | |
| imports <i>Main</i> | | |
| 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 *tranclp*

lemma *tranclp-mono-explicit*:

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

using *rtranclp-mono* **by** (*auto dest!*: *tranclpD intro: rtranclp-into-tranclp2*)

lemma *tranclp-mono*:

assumes *mono*: $r \leq s$

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

using *rtranclp-mono[OF mono]* *mono* **by** (*auto dest!*: *tranclpD intro: rtranclp-into-tranclp2*)

lemma *tranclp-idemp-rel*:

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

apply (*rule iffI*)

prefer 2 **apply** *blast*

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

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

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

by *simp*

lemmas *tranclp-idemp[simp]* = *trancl-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 *rtranclp-unfold*: $rtranclp r a b \longleftrightarrow (a = b \vee tranclp r a b)$

by (*meson rtranclp.simps rtranclpD tranclp-into-rtranclp*)

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

by (*metis rtranclp.rtrancl-refl rtranclp-into-tranclp1 tranclp.cases tranclp-into-rtranclp*)

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

by (*meson rtranclp-into-tranclp2 tranclpD*)

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

apply (*rule iffI*)

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

apply (*induction rule: tranclp-induct; auto simp: trancl-into-trancl2*)

done

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

by (*simp add: rtranclp-unfold*)

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

by (*fastforce simp: rtranclp-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
 $P\text{-finite}: \bigwedge U. P U \longrightarrow U \in A$ **and**
 $finite: finite A$ **and**
 $wf: wf R$ **and**
 $trans: trans R$
shows $wf \{(T, S). (P S \wedge P T) \wedge (T, S) \in lexord R\}$
proof (*rule wfP-if-measure2*)
fix $T S$
assume $P: P S \wedge P T$ **and**
 $s\text{-le-}t: (T, S) \in lexord R$
let $?f = \lambda S. \{U. (U, S) \in lexord R \wedge P U \wedge P S\}$
have $?f T \subseteq ?f S$
 using $s\text{-le-}t P lexord\text{-}trans trans$ **by** *auto*
moreover **have** $T \in ?f S$
 using $s\text{-le-}t P$ **by** *auto*
moreover **have** $T \notin ?f T$
 using $s\text{-le-}t$ **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** $card (?f T) < card (?f S)$ **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* :: $'c \times 'a \Rightarrow \text{bool}$ and *x* :: $'c \times 'a$

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

unfolding *x* by *auto*

show *P x*

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

```

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 (because of other if expression).

```

lemma if-0-1-ge-0[simp]:
   $0 < (\text{if } P \text{ then } a \text{ else } (0::\text{nat})) \longleftrightarrow P \wedge 0 < a$ 
  by auto

```

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 $[?i..<\text{Suc}\ ?j] = (\text{if } ?i \leq ?j \text{ then } [?i..<?j] @ [?j] \text{ else } [])$ leads to a case distinction, that we do not want if the condition is not in the context.

```

lemma upt-Suc-le-append:  $\neg i \leq j \implies [i..<\text{Suc}\ j] = []$ 
  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  $\text{take}\ i\ [m..<n] = [m..<m+i]$ 
  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 $\text{length}\ ?xs \leq ?n \implies \text{take}\ ?n\ ?xs = ?xs$. It is close to $?i + ?m \leq ?n \implies \text{take}\ ?m\ [?i..<?n] = [?i..<?i + ?m]$, but seems more general.

```

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

```

```

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

```

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

```

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

```

oops

A more restrictive version holds:

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

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

proof (induct n)

case 0

then show $?case$ by simp

next

case (Suc n) note IH = this

```

have i: insert 0 {Suc 0..Suc n} = insert 0 {Suc 0..n} ∪ {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..i] and ys: ys = [Suc i..j] and zs: zs = [Suc j..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' ↔ (ys = [] ∧ y = c ∧ ys' = [d]) ∨ (ys = [c] ∧ y = d ∧ ys' = [])
by (cases ys; cases ys') auto

```

lemma *lexn2-conv*:

```

([a, b], [c, d]) ∈ lexn r 2 ↔ (a, c) ∈ r ∨ (a = c ∧ (b, d) ∈ 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* $\equiv \{CF\} \cup \{CT\} \cup \{CVar\ x \mid x. True\}$

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* [φ] = *FNot* φ |
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*]: $(c = CT \vee c = CF \vee c = CVar\ v) \implies wf-conn\ c\ []$ |

wf-conn-unary[*simp*]: $c = CNot \implies wf-conn\ c\ [\psi]$ |

wf-conn-binary[*simp*]: $c \in \text{binary-connectives} \implies wf-conn\ c\ (\psi \# \psi' \# [])$

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

lemma *wf-conn-list-decomp[simp]*:
 $wf\text{-}conn\ CT\ l \longleftrightarrow l = []$
 $wf\text{-}conn\ CF\ l \longleftrightarrow l = []$
 $wf\text{-}conn\ (CVar\ x)\ l \longleftrightarrow l = []$
 $wf\text{-}conn\ CNot\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow \xi = [] \wedge \xi' = []$
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*:
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FT \longleftrightarrow (c = CT \wedge l = [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FF \longleftrightarrow (c = CF \wedge l = [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FVar\ x \longleftrightarrow (c = CVar\ x \wedge l = [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FAnd\ a\ b \longleftrightarrow (c = CAnd \wedge l = a\ \#\ b\ \# [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FOr\ a\ b \longleftrightarrow (c = COr \wedge l = a\ \#\ b\ \# [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FEq\ a\ b \longleftrightarrow (c = CEq \wedge l = a\ \#\ b\ \# [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FImp\ a\ b \longleftrightarrow (c = CImp \wedge l = a\ \#\ b\ \# [])$
 $wf\text{-}conn\ c\ l \implies conn\ c\ l = FNot\ a \longleftrightarrow (c = CNot \wedge l = a\ \# [])$
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 $l :: 'v\ propo\ list$


```

assumes conn:  $c \in \text{binary-connectives}$ 
shows  $\text{length } l = 2 \longleftrightarrow \text{wf-conn } c \ l$ 
proof
  assume  $\text{length } l = 2$ 
  thus  $\text{wf-conn } c \ l$  using wf-conn-binary list-length2-decomp using conn by metis
next
  assume  $\text{wf-conn } c \ l$ 
  thus  $\text{length } l = 2$  (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  $\psi :: 'v \ \text{prop}$ 
    case wf-conn-unary
    thus  $?P \ [\psi]$  using conn binary-connectives-def
    using connective.distinct by blast
  next
    fix  $\psi \ \psi' :: 'v \ \text{prop}$ 
    show  $?P \ [\psi, \ \psi']$  by auto
  qed
qed

lemma wf-conn-not-list-length[iff]:
  fixes  $l :: 'v \ \text{prop} \ \text{list}$ 
  shows  $\text{wf-conn } \text{CNot } l \longleftrightarrow \text{length } l = 1$ 
  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  $l :: 'v \ \text{prop} \ \text{list}$  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: conn c l = FNot  $\psi$ 
shows c = 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** *φ* **where**

subformula-refl[simp]: *$\varphi \preceq \varphi$* |

subformula-into-subformula: *$\psi \in \text{set } l \Rightarrow \text{wf-conn } c \ l \Rightarrow \varphi \preceq \psi \Rightarrow \varphi \preceq \text{conn } c \ l$*

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:

shows *b: FNot $\varphi \preceq \psi \Rightarrow \varphi \preceq \psi$*

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

shows $f \preceq \text{conn } c [f, g]$

and $g \preceq \text{conn } c [f, g]$

proof –

have a : $\text{wf-conn } c (f \# [g])$ **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* ψ' *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 φ) 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*} |
vars-of-prop (*FNot* φ) = *vars-of-prop* φ |
vars-of-prop (*FAnd* φ ψ) = *vars-of-prop* φ \cup *vars-of-prop* ψ |
vars-of-prop (*FOr* φ ψ) = *vars-of-prop* φ \cup *vars-of-prop* ψ |
vars-of-prop (*FImp* φ ψ) = *vars-of-prop* φ \cup *vars-of-prop* ψ |
vars-of-prop (*FEq* φ ψ) = *vars-of-prop* φ \cup *vars-of-prop* ψ

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

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

next

case *binary* **note** *c* = *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 *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* = *this*

fix φ :: '*v* *propo*

have *l* = [ψ] **using** *corr* *c* *incl* *split-list* **by** *force*

thus *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* | *R*

We use *nil* instead of ε .

fun *pos* :: '*v* *propo* \Rightarrow *sign* *list* *set* **where**

pos *FF* = {} |

pos *FT* = {} |

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

$\text{pos } (F\text{And } \varphi \ \psi) = \{\square\} \cup \{L \# p \mid p. p \in \text{pos } \varphi\} \cup \{R \# p \mid p. p \in \text{pos } \psi\} \mid$
 $\text{pos } (F\text{Or } \varphi \ \psi) = \{\square\} \cup \{L \# p \mid p. p \in \text{pos } \varphi\} \cup \{R \# p \mid p. p \in \text{pos } \psi\} \mid$
 $\text{pos } (F\text{Eq } \varphi \ \psi) = \{\square\} \cup \{L \# p \mid p. p \in \text{pos } \varphi\} \cup \{R \# p \mid p. p \in \text{pos } \psi\} \mid$
 $\text{pos } (F\text{Imp } \varphi \ \psi) = \{\square\} \cup \{L \# p \mid p. p \in \text{pos } \varphi\} \cup \{R \# p \mid p. p \in \text{pos } \psi\} \mid$
 $\text{pos } (F\text{Not } \varphi) = \{\square\} \cup \{L \# p \mid p. p \in \text{pos } \varphi\}$

lemma *finite-pos*: *finite* (*pos* φ)
by (*induct* φ , *auto*)

lemma *finite-inj-comp-set*:

fixes $s :: 'v \text{ set}$
assumes *finite*: *finite* s
and *inj*: *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 : *finite* s **and** *notin*: $x \notin s$
and *IH*: $\text{card } \{f \ p \mid p. p \in s\} = \text{card } s$
have f' : *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*:

inj (*op* $\#$ s)
by (*meson* *injI* *list.inject*)

lemma *finite-insert-nil-cons*:

finite $s \implies \text{card } (\text{insert } \square \ \{L \# p \mid p. p \in s\}) = 1 + \text{card } \{L \# p \mid p. p \in s\}$
using *card-insert-disjoint* **by** *auto*

lemma *card-not[simp]*:

$\text{card } (\text{pos } (F\text{Not } \varphi)) = 1 + \text{card } (\text{pos } \varphi)$

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

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

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

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

unfolding *prop-size-def* **apply** (*induct* φ , *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 *pos* (*FImp* (*FAnd* (*FVar* *P*) (*FVar* *Q*)) (*FOr* (*FVar* *P*) (*FVar* *Q*)))

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

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

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

$\implies \text{path-to } (L \# p) \ (\text{conn } c \ (\varphi \# l)) \ \varphi' \mid$

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 *path-to-subformula*:

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 *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 *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  $wf = \text{this}(2)$  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  $\exists p. \text{path-to } p \ (\text{conn } c \ l) \ \varphi'$  by blast
}
moreover {
  assume  $c: c = CNot$ 
  hence  $l = [\psi]$  using  $wf \ \psi \ \text{wf-conn-Not-decomp}$  by fastforce
  hence  $\text{path-to } (L \ \# \ p) \ (\text{conn } c \ l) \ \varphi'$  by (metis  $c \ \text{wf-conn-unary } p \ \text{path-to-l}$ )
  hence  $\exists p. \text{path-to } p \ (\text{conn } c \ l) \ \varphi'$  by blast
}
moreover {
  assume  $c: c \in \text{binary-connectives}$ 
  obtain  $a \ b$  where  $ab: [a, b] = l$  using subformula-into-subformula  $c \ \text{wf-conn-bin-list-length}$ 
  list-length2-decomp by metis
  hence  $a = \psi \vee b = \psi$  using  $\psi$  by auto
  hence  $\text{path-to } (L \ \# \ p) \ (\text{conn } c \ l) \ \varphi' \vee \text{path-to } (R \ \# \ p) \ (\text{conn } c \ l) \ \varphi'$  using  $c \ \text{path-to-l}$ 
  path-to-r  $p \ ab$  by (metis wf-conn-binary)
  hence  $\exists p. \text{path-to } p \ (\text{conn } c \ l) \ \varphi'$  by blast
}
ultimately show  $\exists p. \text{path-to } p \ (\text{conn } c \ l) \ \varphi'$  using connective-cases-arity by metis
qed

```

```

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

```

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

```


definition *evalf* (**infix** \models_f 50) **where**
evalf $\varphi \psi = (\forall A. A \models \varphi \longrightarrow A \models \psi)$

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

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 φ entails ψ (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 φ entails ψ , and therefore $A \models FImp \varphi \psi$.

assume $A \models \varphi$
hence $A \models \psi$ **using** H **unfolding** *evalf-def* **by** *metis*
hence $A \models FImp \varphi \psi$ **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 $\neg A \models \varphi$
hence $A \models FImp \varphi \psi$ **by** *auto*
}

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

ultimately have $A \models FImp \varphi \psi$ **by** *blast*

}

thus $\forall A. A \models FImp \varphi \psi$ **by** *blast*

next

show $\forall A. A \models FImp \varphi \psi \implies \varphi \models_f \psi$

proof (*rule ccontr*)

assume $\neg \varphi \models_f \psi$

then obtain A **where** $A \models \varphi \wedge \neg A \models \psi$ **using** *evalf-def* **by** *metis*

hence $\neg A \models FImp \varphi \psi$ **by** *auto*

moreover assume $\forall A. A \models FImp \varphi \psi$

ultimately show *False* **by** *blast*

qed

qed

A shorter proof:

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

by (*simp add: evalf-def*)

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

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

lemma *same-over-set-eval*:

```

assumes same-over-set  $A\ B$  (vars-of-prop  $\varphi$ )
shows  $A \models \varphi \longleftrightarrow B \models \varphi$ 
using assms unfolding same-over-set-def by (induct  $\varphi$ , 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:  $r\ \varphi\ \psi \Longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \psi$  |
    propo-rew-one-step-lift:  $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow wf\text{-}conn\ c\ (\psi s\ @\ \varphi\ \# \psi s') \Longrightarrow propo\text{-}rew\text{-}step\ r\ (conn\ c\ (\psi s\ @\ \varphi\ \# \psi s'))\ (conn\ c\ (\psi s\ @\ \varphi' \# \psi s'))$ 

```

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

This lemma is only a health condition:

```

lemma propo-rew-step-subformula-imp:
shows  $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$ 
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 ψ and ψ' , then every formula φ containing ψ , can be rewritten into a formula φ' , such that it contains φ' .

```

lemma propo-rew-step-subformula-rec:
  fixes  $\psi\ \psi'\ \varphi :: 'v\ propo$ 
  shows  $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists\ \varphi'.\ \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi')$ 
proof (induct  $\varphi$  rule: subformula.induct)
  case subformula-refl
  hence  $propo\text{-}rew\text{-}step\ r\ \psi\ \psi'$  using propo-rew-step.intros by auto
  moreover have  $\psi' \preceq \psi'$  using Prop-Logic.subformula-refl by auto
  ultimately show  $\exists\ \varphi'.\ \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \psi\ \varphi'$  by fastforce
next
  case (subformula-into-subformula  $\psi''\ l\ c$ )
  note  $IH = this(4)$  and  $r = this(5)$  and  $\psi'' = this(1)$  and  $wf = this(2)$  and  $incl = this(3)$ 
  then obtain  $\varphi'$  where  $*: \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \psi''\ \varphi'$  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-conn-no-arity-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* $c = \text{this}$
 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* $c = \text{this}$
 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: *propo-rew-step* $r \varphi \varphi'$ means that we rewrite ψ inside φ (ie at a path p) into ψ' .

lemma *propo-rew-step-rewrite*:
 fixes $\varphi \varphi' :: 'v \text{ propo}$ **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* $\square \varphi \varphi$ **by** *auto*
 moreover have *replace-at* $\square \varphi \psi = \psi$ **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 ∨ c = CF ∨ c = CVar x
hence False using corr by auto
hence ∃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'))
  by fast
}
moreover {
  assume c: c = CNot
  hence empty: ξ = [] ξ' = [] 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-neq*)

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 $f \implies \text{preserves-un-sat } g \implies \text{preserves-un-sat } (f \text{ OO } g)$

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 $r \implies \text{preserves-un-sat } (\text{propo-rew-step } r)^{**}$

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

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

lemma *test-symb-imp-all-subformula-st[simp]*:
test-symb FT \Longrightarrow *all-subformula-st test-symb FT*
test-symb FF \Longrightarrow *all-subformula-st test-symb FF*
test-symb (FVar x) \Longrightarrow *all-subformula-st test-symb (FVar x)*
unfolding *all-subformula-st-def* **using** *subformula-leaf* **by** *metis+*

lemma *all-subformula-st-test-symb-true-phi*:
all-subformula-st 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 $\Longrightarrow (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
 \Longrightarrow *all-subformula-st 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 test-symb (conn c l) $\Longrightarrow \text{wf-conn } c \ l$
 $\Longrightarrow (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
unfolding *all-subformula-st-def* **by** *auto*

lemma *all-subformula-st-decomp*:
fixes *c* :: 'v connective **and** *l* :: 'v propo list
assumes *wf-conn c l*
shows *all-subformula-st test-symb (conn c l)*
 $\longleftrightarrow (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
using *assms all-subformula-st-decomp-rec all-subformula-st-decomp-imp* **by** *metis*

lemma *helper-fact*: *c* \in *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 propo
shows *all-subformula-st test-symb (FAnd $\varphi \ \psi$)*
 $\longleftrightarrow (\text{test-symb } (FAnd \ \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb (FOr $\varphi \ \psi$)*
 $\longleftrightarrow (\text{test-symb } (FOr \ \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb (FNot φ)*
 $\longleftrightarrow (\text{test-symb } (FNot \ \varphi) \wedge \text{all-subformula-st test-symb } \varphi)$
and *all-subformula-st test-symb (FEq $\varphi \ \psi$)*
 $\longleftrightarrow (\text{test-symb } (FEq \ \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb (FImp $\varphi \ \psi$)*
 $\longleftrightarrow (\text{test-symb } (FImp \ \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$

proof –

have *all-subformula-st test-symb* (*FAnd* φ ψ) \longleftrightarrow *all-subformula-st test-symb* (*conn CAnd* $[\varphi, \psi]$)
by *auto*
moreover have $\dots \longleftrightarrow$ *test-symb* (*conn CAnd* $[\varphi, \psi]$) \wedge ($\forall \xi \in \text{set } [\varphi, \psi]. \text{all-subformula-st test-symb } \xi$)
using *all-subformula-st-decomp wf-conn-helper-facts(5)* **by** *metis*
finally show *all-subformula-st test-symb* (*FAnd* φ ψ)
 \longleftrightarrow (*test-symb* (*FAnd* φ ψ) \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
by *simp*

have *all-subformula-st test-symb* (*FOr* φ ψ) \longleftrightarrow *all-subformula-st test-symb* (*conn COr* $[\varphi, \psi]$)
by *auto*
moreover have $\dots \longleftrightarrow$
(*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* φ ψ)
 \longleftrightarrow (*test-symb* (*FOr* φ ψ) \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
by *simp*

have *all-subformula-st test-symb* (*FEq* φ ψ) \longleftrightarrow *all-subformula-st test-symb* (*conn CEq* $[\varphi, \psi]$)
by *auto*
moreover have \dots
 \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* φ ψ)
 \longleftrightarrow (*test-symb* (*FEq* φ ψ) \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
by *simp*

have *all-subformula-st test-symb* (*FImp* φ ψ) \longleftrightarrow *all-subformula-st test-symb* (*conn CImp* $[\varphi, \psi]$)
by *auto*
moreover have \dots
 \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* φ ψ)
 \longleftrightarrow (*test-symb* (*FImp* φ ψ) \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
by *simp*

have *all-subformula-st test-symb* (*FNot* φ) \longleftrightarrow *all-subformula-st test-symb* (*conn CNot* $[\varphi]$)
by *auto*
moreover have $\dots =$ (*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* φ)
 \longleftrightarrow (*test-symb* (*FNot* φ) \wedge *all-subformula-st test-symb* φ) **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 φ .

lemma *no-test-symb-step-exists*:

```

fixes r:: 'v propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool and test-symb:: 'v propo  $\Rightarrow$  bool and x :: 'v
and  $\varphi$  :: '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 = this(1)[OF this(2)] and r = this(2) and nst = this(3) and subf =
this(4)
from r 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 = this(4)
and  $\varphi = \text{this}(3)$  and le = this(5) and nst = 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:  $\text{wf-conn } c\ [\varphi1, \varphi2]$  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 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 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 $\varphi1$  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: $\forall c \xi \varphi \xi' \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \varphi \varphi' \longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$

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 Φ will moreover hold for the subterm we are rewriting. For example if there is no equivalence symbol in Φ , we do not have to care about equivalence symbols in the two previous assumptions.

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

```

fixes  $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  and  $\text{test-symb} :: 'v \text{ propo} \Rightarrow \text{bool}$  and  $x :: 'v$ 
and  $\varphi \psi \Phi :: 'v \text{ propo}$ 
assumes  $H: \forall \varphi' \psi. \varphi' \preceq \Phi \longrightarrow r \varphi' \psi \longrightarrow \text{all-subformula-st test-symb } \varphi'$ 
 $\longrightarrow \text{all-subformula-st test-symb } \psi$ 
and  $H': \forall (c :: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \varphi \varphi'$ 
 $\longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi'$ 
 $\longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$  and
 $\text{propo-rew-step } r \varphi \psi$  and
 $\varphi \preceq \Phi$  and
 $\text{all-subformula-st test-symb } \varphi$ 
shows  $\text{all-subformula-st test-symb } \psi$ 
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  $\text{rel} = \text{this}(1)$  and  $\varphi = \text{this}(2)$  and  $\text{corr} = \text{this}(3)$  and  $\Phi = \text{this}(4)$  and  $\text{nst} = \text{this}(5)$ 
have  $\text{sq}: \varphi \preceq \Phi$ 
using  $\Phi \text{ corr subformula-into-subformula subformula-refl subformula-trans}$ 
by (metis in-set-conv-decomp)
from  $\text{corr}$  have  $\forall \psi. \psi \in \text{set } (\xi @ \varphi \# \xi') \longrightarrow \text{all-subformula-st test-symb } \psi$ 
using  $\text{all-subformula-st-decomp nst}$  by blast
hence *:  $\forall \psi. \psi \in \text{set } (\xi @ \varphi' \# \xi') \longrightarrow \text{all-subformula-st test-symb } \psi$  using  $\varphi \text{ sq}$  by fastforce
hence  $\text{test-symb } \varphi'$  using  $\text{all-subformula-st-test-symb-true-phi}$  by auto
moreover from  $\text{corr nst}$  have  $\text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi'))$ 
using  $\text{all-subformula-st-decomp}$  by blast
ultimately have  $\text{test-symb: test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$  using  $H' \text{ sq corr rel}$  by blast

have  $\text{wf-conn } c (\xi @ \varphi' \# \xi')$ 
by (metis wf-conn-no-arity-change-helper corr wf-conn-no-arity-change)
thus  $\text{all-subformula-st test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
using *  $\text{test-symb}$  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 $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **and** *test-symb* :: $'v \text{ propo} \Rightarrow \text{bool}$ **and** $x :: 'v$
and $\varphi \psi :: 'v \text{ propo}$

assumes

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

propo-rew-step $r \varphi \psi$ **and**

all-subformula-st test-symb φ

shows *all-subformula-st test-symb* ψ

using *propo-rew-step-inv-stay* [of $\varphi \ r \ \text{test-symb } \varphi \ \psi$] *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 $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **and** *test-symb* :: $'v \text{ propo} \Rightarrow \text{bool}$ **and** $x :: 'v$
and $\varphi \psi :: 'v \text{ propo}$

assumes

$H: \forall \varphi \psi. \text{propo-rew-step } r \varphi \psi \longrightarrow \text{all-subformula-st test-symb } \varphi$
 $\longrightarrow \text{all-subformula-st test-symb } \psi$ **and**

$H': \forall (c :: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \varphi \varphi'$
 $\longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi'$
 $\longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ **and**

$\varphi \preceq \Phi$ **and**

full: *full* (*propo-rew-step* r) $\varphi \psi$ **and**

init: *all-subformula-st test-symb* φ

shows *all-subformula-st test-symb* ψ

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* ψ

using *init*

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

case *base*

then show *all-subformula-st test-symb* φ **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'* $H \ H' \ \text{rel } \text{one}$ **by** *auto*

qed

qed

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

fixes $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **and** *test-symb* :: $'v \text{ propo} \Rightarrow \text{bool}$ **and** $x :: 'v$

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

assumes

$H: \forall \varphi \psi. \text{propo-rew-step } r \varphi \psi \longrightarrow \text{all-subformula-st test-symb } \varphi$
 $\longrightarrow \text{all-subformula-st test-symb } \psi$ **and**

$H': \forall (c :: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{propo-rew-step } r \varphi \varphi' \longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi')$
 $\longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ **and**

full: *full* (*propo-rew-step* r) $\varphi \psi$ **and**

init: *all-subformula-st test-symb* φ

shows *all-subformula-st test-symb* ψ

using *full-propo-rew-step-inv-stay-with-inc*[of *r test-symb* φ] *assms subformula-refl* **by** *metis*

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

fixes $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **and** *test-symb* :: $'v \text{ propo} \Rightarrow \text{bool}$ **and** $x :: 'v$

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

assumes

$H: \forall \varphi \psi. r \varphi \psi \longrightarrow \text{all-subformula-st test-symb } \varphi \longrightarrow \text{all-subformula-st test-symb } \psi$ **and**

$H': \forall (c :: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{wf-conn } c (\xi @ \varphi \# \xi') \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi'))$
 $\longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ **and**

full: *full* (*propo-rew-step* *r*) $\varphi \psi$ **and**

init: *all-subformula-st test-symb* φ

shows *all-subformula-st test-symb* ψ

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* ψ

using *init*

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

case *base*

thus *all-subformula-st test-symb* φ **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

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

fixes $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **and** *test-symb* :: $'v \text{ propo} \Rightarrow \text{bool}$ **and** $x :: 'v$

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

assumes

$H: \forall \varphi \psi. r \varphi \psi \longrightarrow \text{all-subformula-st test-symb } \varphi \longrightarrow \text{all-subformula-st test-symb } \psi$ **and**

$H': \forall (c :: 'v \text{ connective}) l l'. \text{wf-conn } c l \longrightarrow \text{wf-conn } c l'$

$\longrightarrow (\text{test-symb } (\text{conn } c l) \longleftrightarrow \text{test-symb } (\text{conn } c l'))$ **and**

full: *full* (*propo-rew-step* *r*) $\varphi \psi$ **and**

init: *all-subformula-st test-symb* φ

shows *all-subformula-st test-symb* ψ

proof –

have $\bigwedge (c :: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{wf-conn } c (\xi @ \varphi \# \xi')$

$\implies \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \implies \text{test-symb } \varphi' \implies \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$

using *H'* **by** (*metis wf-conn-no-arity-change-helper wf-conn-no-arity-change*)

thus *all-subformula-st test-symb* ψ

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 φ ψ) (FAnd (FImp φ ψ) (FImp ψ φ))

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-consistent*:
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
 $\neg \text{no-equiv } (\text{FEq } \varphi \ \psi)$
no-equiv FT
no-equiv 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 $\varphi \ \psi :: 'v \text{ propo}$

shows

$\text{no-equiv } (F\text{Not } \varphi) \longleftrightarrow \text{no-equiv } \varphi$
 $\text{no-equiv } (F\text{And } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$
 $\text{no-equiv } (F\text{Or } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$
 $\text{no-equiv } (F\text{Imp } \varphi \ \psi) \longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$
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*:

$\forall x::'v. \text{no-equiv-symb } FF \wedge \text{no-equiv-symb } FT \wedge \text{no-equiv-symb } (F\text{Var } x)$

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 $\bigwedge p \ \text{pa}. \neg \text{elim-equiv } (p::'v \text{ propo}) \ \text{pa} \vee \neg \text{no-equiv-symb } p$

using *elim-equiv.cases no-equiv-symb.simps(1)* **by** *blast*

hence $\text{elim-equiv } (\text{conn } c \ l) \ \psi \implies \neg \text{no-equiv-symb } (\text{conn } c \ l) \ \text{using } a1 \text{ by } \textit{metis}$

}

moreover **have** $H': \forall \psi. \neg \text{elim-equiv } FT \ \psi \ \forall \psi. \neg \text{elim-equiv } FF \ \psi \ \forall \psi \ x. \neg \text{elim-equiv } (F\text{Var } x) \ \psi$

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* φ , *auto simp add: elim-equiv.simps*)

hence $\bigwedge \varphi'. \ \varphi' \preceq \varphi \implies \neg \text{no-equiv-symb } \varphi' \implies \exists \psi. \text{elim-equiv } \varphi' \ \psi \text{ by } \textit{force}$

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* φ) ψ)

lemma *elim-imp-transformation-consistent*:

$A \models F\text{Imp } \varphi \ \psi \longleftrightarrow A \models F\text{Or } (F\text{Not } \varphi) \ \psi$

by *auto*

lemma *elim-imp-explicit*: $\text{elim-imp } \varphi \ \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$

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 φ ψ)*
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 φ) \longleftrightarrow no-imp φ
no-imp (FAnd $\varphi \psi$) \longleftrightarrow (no-imp $\varphi \wedge$ no-imp ψ)
no-imp (FOr $\varphi \psi$) \longleftrightarrow (no-imp $\varphi \wedge$ no-imp ψ)
by *auto*

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv:*
elim-imp $\varphi \psi \implies$ no-equiv $\varphi \implies$ no-equiv ψ
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 φ*
shows *no-equiv ψ*
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 φ*
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  $(\bigwedge \varphi'. \varphi' \preceq \varphi \implies \neg$ 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) $\varphi \psi \implies$ no-imp ψ
 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 φ FT) φ |

ElimTB1': *elimTB* (FAnd FT φ) φ |

ElimTB2: *elimTB* (FAnd φ FF) FF |

ElimTB2': *elimTB* (FAnd FF φ) FF |

ElimTB3: *elimTB* (FOr φ FT) FT |

ElimTB3': *elimTB* (FOr FT φ) FT |

ElimTB4: *elimTB* (FOr φ FF) φ |

ElimTB4': *elimTB* (FOr FF φ) φ |

ElimTB5: *elimTB* (FNot FT) FF |

ElimTB6: *elimTB* (FNot FF) FT

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

proof –

```

{
  fix  $\varphi \psi$ :: 'b 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 no-T-F-symb (conn c l)

lemma *wf-conn-no-T-F-symb-iff*[simp]:

$\text{wf-conn } c \ \psi s \implies \text{no-T-F-symb (conn } c \ \psi s) \longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in \text{set } \psi s. \psi \neq FF \wedge \psi \neq$

```

FT))
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
   $\neg$ no-T-F-symb (FT :: 'v propo)
   $\neg$ no-T-F-symb (FF :: 'v 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:
 $\neg$ no-T-F-symb (FNot  $\varphi$ )  $\implies \varphi = FT \vee \varphi = FF$ 
proof (rule ccontr)
  assume n:  $\neg$  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]:
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* $\varphi \implies$ *no-T-F-symb-except-toplevel* φ

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*:
 $\varphi \neq FT \implies \varphi \neq FF \implies$ *no-T-F-symb-except-toplevel* (*FNot* φ)
by *simp*

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes $\varphi \ \psi$:: 'v *propo*
assumes $\varphi \neq FT$ **and** $\varphi \neq FF$ **and** $\psi \neq FT$ **and** $\psi \neq FF$
and *c*: *c* ∈ *binary-connectives*
shows *no-T-F-symb-except-toplevel* (*conn* *c* [φ , ψ])
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 *propo* *list* **and** *c* :: 'v *connective*
assumes *corr*: *wf-conn* *c* *l*
and *FT* ∈ *set* *l* ∨ *FF* ∈ *set* *l*
shows \neg *no-T-F-symb-except-toplevel* (*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 *propo*
assumes $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
shows
 \neg *no-T-F-symb-except-toplevel* (*FAnd* $\varphi \ \psi$)
 \neg *no-T-F-symb-except-toplevel* (*FOr* $\varphi \ \psi$)
 \neg *no-T-F-symb-except-toplevel* (*FImp* $\varphi \ \psi$)
 \neg *no-T-F-symb-except-toplevel* (*FEq* $\varphi \ \psi$)
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 *propo*
assumes $\varphi = FT \vee \varphi = FF$
shows
 \neg *no-T-F-symb-except-toplevel* (*FNot* φ)
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 *all-subformula-st no-T-F-symb-except-toplevel*

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

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

no-T-F \equiv *all-subformula-st no-T-F-symb*

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

fixes *l* :: 'v propo list **and** *c* :: 'v connective

assumes *wf-conn c l*

and *FT* \in *set l* \vee *FF* \in *set l*

shows \neg *no-T-F-except-top-level* (*conn c l*)

by (*simp add: all-subformula-st-decomp assms no-T-F-except-top-level-def no-T-F-symb-except-toplevel-if-is-a-true-false*)

lemma *no-T-F-except-top-level-false-example*[*simp*]:

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

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

shows

\neg *no-T-F-except-top-level* (*FAnd* $\varphi \psi$)

\neg *no-T-F-except-top-level* (*FOr* $\varphi \psi$)

\neg *no-T-F-except-top-level* (*FEq* $\varphi \psi$)

\neg *no-T-F-except-top-level* (*FImp* $\varphi \psi$)

by (*metis all-subformula-st-test-symb-true-phi assms no-T-F-except-top-level-def no-T-F-symb-except-top-level-false-example*)**+**

lemma *no-T-F-symb-except-toplevel-no-T-F-symb*:

no-T-F-symb-except-toplevel $\varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies$ *no-T-F-symb* φ

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-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*:

no-T-F-except-top-level $\varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies$ *no-T-F* φ

unfolding *no-T-F-except-top-level-def no-T-F-def* **apply** (*induct* φ)

using *no-T-F-symb-fnot* **by** *fastforce***+**

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

no-T-F $\varphi \implies$ *no-T-F-except-top-level* φ

unfolding *no-T-F-except-top-level-def no-T-F-def*

unfolding *all-subformula-st-def* **by** *auto*

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

unfolding *no-T-F-except-top-level-def* **by** *auto*

lemma *no-T-F-no-T-F-except-top-level'*[*simp*]:

no-T-F-except-top-level $\varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee$ *no-T-F* $\varphi)$

apply *auto*

using *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-no-T-F-except-top-level*

by *blast***+**

lemma *no-T-F-bin-decomp*[*simp*]:

assumes *c*: *c* \in *binary-connectives*

shows $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
proof –
have $\text{wf: wf-conn } c \ [\varphi, \psi]$ **using** c **by** *auto*
hence $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F-symb } (\text{conn } c \ [\varphi, \psi]) \wedge \text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
by (*simp add: all-subformula-st-decomp no-T-F-def*)
thus $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
using c *wf all-subformula-st-decomp list.discI no-T-F-def no-T-F-symb-except-toplevel-bin-decom*
 $\text{no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(1,2) wf-conn-helper-facts(2,3)}$
 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 $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
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
 $\text{no-T-F } (FAnd \ \varphi \ \psi) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
 $\text{no-T-F } (FOr \ \varphi \ \psi) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
 $\text{no-T-F } (FEq \ \varphi \ \psi) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
 $\text{no-T-F } (FImp \ \varphi \ \psi) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
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 $\text{no-T-F } (FNot \ \varphi) \longleftrightarrow \text{no-T-F } \varphi$
by (*metis all-subformula-st-decomp-explicit(3) all-subformula-st-test-symb-true-phi no-T-F-def*
 $\text{no-T-F-symb-false(1,2) no-T-F-symb-fnot-imp}$)

lemma *no-T-F-decomp:*
fixes $\varphi \ \psi :: 'v \text{ propo}$
assumes $\varphi: \text{no-T-F } (FAnd \ \varphi \ \psi) \vee \text{no-T-F } (FOr \ \varphi \ \psi) \vee \text{no-T-F } (FEq \ \varphi \ \psi) \vee \text{no-T-F } (FImp \ \varphi \ \psi)$
shows $\text{no-T-F } \psi$ **and** $\text{no-T-F } \varphi$
using *assms* **by** *auto*

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

lemma *no-T-F-symb-except-toplevel-step-exists:*
fixes $\varphi \ \psi :: 'v \text{ propo}$
assumes $\text{no-equiv } \varphi$ **and** $\text{no-imp } \varphi$
shows $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTB } \psi \ \psi'$
proof (*induct ψ 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 ψ*)
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  $\varphi' \psi1 \psi2$ )
note IH1 = this(1) and IH2 = this(2) and  $\varphi' = this(3)$  and  $F\varphi = this(4)$  and  $n = this(5)$ 
{
  assume  $\varphi' = FImp \psi1 \psi2 \vee \varphi' = FEq \psi1 \psi2$ 
  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:  $\neg$  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  $c:: 'v \text{ connective}$  and  $l:: 'v \text{ propo list}$  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:  $\text{elimTB.induct}$ , auto)
}
thus  $\text{no-equiv } \psi$ 
  using  $\text{full-propo-rew-step-inv-stay-conn[of elimTB no-equiv-symb } \varphi \psi]$ 
     $\text{no-equiv-symb-conn-characterization assms unfolding no-equiv-def by metis}$ 
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  $\text{elimTB-full-propo-rew-step}$ :
  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**

```

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

```

lemma $\text{pushNeg-transformation-consistent}$:

```

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

```

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

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

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

fun simple **where**

```

 $\text{simple FT} = \text{True} |$ 
 $\text{simple FF} = \text{True} |$ 
 $\text{simple (FVar -)} = \text{True} |$ 

```

simple - = *False*

lemma *simple-decomp*:

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

lemma *subformula-conn-decomp-simple*:

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

assumes *s*: *simple* ψ

shows $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \vee \varphi = \psi)$

proof -

have $\varphi \preceq conn\ CNot\ [\psi] \longleftrightarrow (\varphi = conn\ CNot\ [\psi] \vee (\exists \psi \in set\ [\psi]. \varphi \preceq \psi))$

using *subformula-conn-decomp wf-conn-helper-facts(1)* **by** *metis*

thus $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \vee \varphi = \psi)$ **using** *s* **by** (*auto simp add: simple-decomp*)

qed

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

fixes $\varphi :: 'v\ propo$ **and** $x :: 'v$

shows

$\varphi \preceq FNot\ FT \longleftrightarrow (\varphi = FNot\ FT \vee \varphi = FT)$

$\varphi \preceq FNot\ FF \longleftrightarrow (\varphi = FNot\ FF \vee \varphi = FF)$

$\varphi \preceq FNot\ (FVar\ x) \longleftrightarrow (\varphi = FNot\ (FVar\ x) \vee \varphi = FVar\ x)$

by (*auto simp add: subformula-conn-decomp-simple*)

fun *simple-not-symb* **where**

simple-not-symb (*FNot* φ) = (*simple* φ) |

simple-not-symb - = *True*

definition *simple-not* **where**

simple-not = *all-subformula-st simple-not-symb*

declare *simple-not-def[simp]*

lemma *simple-not-Not[simp]*:

$\neg simple-not\ (FNot\ (FAnd\ \varphi\ \psi))$

$\neg simple-not\ (FNot\ (FOr\ \varphi\ \psi))$

by *auto*

lemma *simple-not-step-exists*:

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

assumes *no-equiv* φ **and** *no-imp* φ

shows $\psi \preceq \varphi \implies \neg simple-not-symb\ \psi \implies \exists \psi'. pushNeg\ \psi\ \psi'$

apply (*induct* ψ , *auto*)

apply (*case-tac* ψ , *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 $\varphi :: 'v\ propo$

assumes *noTB*: $\neg simple-not\ \varphi$ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ

shows $\exists \psi\ \psi'. \psi \preceq \varphi \wedge pushNeg\ \psi\ \psi'$

proof -

have $\forall x. simple-not-symb\ (FF :: 'v\ propo) \wedge simple-not-symb\ FT \wedge simple-not-symb\ (FVar\ (x :: 'v))$

by *auto*

moreover {

```

    fix c:: 'v connective and l :: 'v propo list and  $\psi :: 'v propo$ 
    have H: pushNeg (conn c l)  $\psi \implies \neg \text{simple-not-symb (conn c l)}$ 
      by (case-tac (conn c l) rule: pushNeg.cases, simp-all)
  }
  moreover {
    fix x :: 'v
    have H': simple-not FT simple-not FF simple-not (FVar x)
      by simp-all
  }
  moreover {
    fix  $\psi :: 'v propo$ 
    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:
  no-T-F-except-top-level (FNot (FAnd  $\varphi \psi$ ))  $\implies$  no-T-F-except-top-level (FOr (FNot  $\varphi$ ) (FNot  $\psi$ ))
  using no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-comp-not no-T-F-decomp(1)
    no-T-F-decomp(2) no-T-F-no-T-F-except-top-level by (metis no-T-F-comp-expanded-explicit(2)
      propo.distinct(5,17))

lemma no-T-F-except-top-level-pushNeg2:
  no-T-F-except-top-level (FNot (FOr  $\varphi \psi$ ))  $\implies$  no-T-F-except-top-level (FAnd (FNot  $\varphi$ ) (FNot  $\psi$ ))
  by auto

lemma no-T-F-symb-pushNeg:
  no-T-F-symb (FOr (FNot  $\varphi'$ ) (FNot  $\psi'$ ))
  no-T-F-symb (FAnd (FNot  $\varphi'$ ) (FNot  $\psi'$ ))
  no-T-F-symb (FNot (FNot  $\varphi'$ ))
  by auto

lemma propo-rew-step-pushNeg-no-T-F-symb:
  propo-rew-step pushNeg  $\varphi \psi \implies$  no-T-F-except-top-level  $\varphi \implies$  no-T-F-symb  $\varphi \implies$  no-T-F-symb  $\psi$ 
  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  $\varphi \varphi':: 'a propo$  and  $c:: 'a connective$  and  $\xi \xi':: 'a propo list$ 
  assume rel: propo-rew-step pushNeg  $\varphi \varphi'$ 
  and IH: no-T-F  $\varphi \implies$  no-T-F-symb  $\varphi \implies$  no-T-F-symb  $\varphi'$ 
  and wf: wf-conn c ( $\xi @ \varphi \# \xi'$ )
  and n: conn c ( $\xi @ \varphi \# \xi'$ ) = FF  $\vee$  conn c ( $\xi @ \varphi \# \xi'$ ) = FT  $\vee$  no-T-F (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  $n': \text{no-T-F (conn c } (\xi @ \varphi \# \xi'))$  using n by (simp add: wf wf-conn-list(1,2))
  moreover
  {
    have 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 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 no-T-F-symb (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 no-T-F \varphi \implies 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  $c :: 'v$  connective and  $\xi \xi' :: 'v$  propo list and  $\zeta \zeta' :: 'v$  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  $c (\xi @ \zeta \# \xi')$ )
  and n: no-T-F-symb-except-toplevel  $\zeta'$ 
  have no-T-F-symb-except-toplevel (conn  $c (\xi @ \zeta' \# \xi')$ )
  proof
    have p: no-T-F-symb (conn  $c (\xi @ \zeta \# \xi')$ )
    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:  $\forall \varphi \in \text{set } (\xi @ \zeta \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$ 
    using corr wf-conn-no-T-F-symb-iff p by blast
    from rel incl have  $\zeta' \neq FT \wedge \zeta' \neq FF$ 
    apply (induction  $\zeta \zeta'$  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  $\forall \varphi \in \text{set } (\xi @ \zeta' \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$  using l by auto
    moreover have  $c \neq CT \wedge c \neq CF$  using corr by auto
    ultimately show no-T-F-symb (conn  $c (\xi @ \zeta' \# \xi')$ )
    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  $\psi$ 
using full-propo-rew-step-inv-stay-with-inc[of pushNeg no-T-F-symb-except-toplevel  $\varphi$ ] assms
subformula-refl unfolding no-T-F-except-top-level-def full-unfold by metis
next
{
  fix  $\varphi \psi :: 'v$  propo
  have H: pushNeg  $\varphi \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$ 
  by (induct  $\varphi \psi$  rule: pushNeg.induct, auto)
}
thus no-equiv  $\psi$ 
using full-propo-rew-step-inv-stay-conn[of pushNeg no-equiv-symb  $\varphi \psi$ ]
no-equiv-symb-conn-characterization assms unfolding no-equiv-def full-unfold by metis
next
{
  fix  $\varphi \psi :: 'v$  propo
  have H: pushNeg  $\varphi \psi \implies \text{no-imp } \varphi \implies \text{no-imp } \psi$ 
  by (induct  $\varphi \psi$  rule: pushNeg.induct, auto)
}
thus no-imp  $\psi$ 
using full-propo-rew-step-inv-stay-conn[of pushNeg no-imp-symb  $\varphi \psi$ ] assms
no-imp-symb-conn-characterization unfolding no-imp-def full-unfold by metis
qed

```

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

fixes $\varphi \psi :: 'v \text{ propo}$
assumes
 $\text{no-equiv } \varphi$ **and**
 $\text{no-imp } \varphi$ **and**
 $\text{full } (\text{propo-rew-step } \text{pushNeg}) \varphi \psi$ **and**
 $\text{no-T-F-except-top-level } \varphi$
shows $\text{simple-not } \psi$
using $\text{assms full-propo-rew-step-subformula pushNeg-inv}(1,2) \text{ simple-not-rew by blast}$

8.5 Push inside

inductive $\text{push-conn-inside} :: 'v \text{ connective} \Rightarrow 'v \text{ connective} \Rightarrow 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$

for $c \ c' :: 'v \text{ connective}$ **where**

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

$\Longrightarrow \text{push-conn-inside } c \ c' (\text{conn } c [\text{conn } c' [\varphi 1, \varphi 2], \psi])$
 $(\text{conn } c' [\text{conn } c [\varphi 1, \psi], \text{conn } c [\varphi 2, \psi]]) \mid$

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

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

lemma $\text{push-conn-inside-explicit}: \text{push-conn-inside } c \ c' \varphi \psi \Longrightarrow \forall A. A \models \varphi \longleftrightarrow A \models \psi$
by ($\text{induct } \varphi \psi \text{ rule: push-conn-inside.induct, auto}$)

lemma $\text{push-conn-inside-consistent}: \text{preserves-un-sat } (\text{push-conn-inside } c \ c')$
unfolding $\text{preserves-un-sat-def}$ **by** ($\text{simp add: push-conn-inside-explicit}$)

lemma $\text{propo-rew-step-push-conn-inside[simp]}:$

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

proof –

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

$\}$

thus

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

qed

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

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

$\Longrightarrow \text{not-c-in-c'-symb } c \ c' (\text{conn } c [\text{conn } c' [\varphi, \varphi'], \psi]) \mid$

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

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

abbreviation $c\text{-in-c'-symb } c \ c' \varphi \equiv \neg \text{not-c-in-c'-symb } c \ c' \varphi$

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

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

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]*:

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

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

$\neg not-c-in-c'-symb\ c\ c'\ (FVar\ x)$

$\neg not-c-in-c'-symb\ c\ c'\ (FNot\ FF)$

$\neg not-c-in-c'-symb\ c\ c'\ (FNot\ FT)$

$\neg not-c-in-c'-symb\ c\ c'\ (FNot\ (FVar\ x))$

using *c-in-c'-symb-simp* **by** *metis+*

definition *c-in-c'-only* **where**

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

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

c-in-c'-only c c' FF

c-in-c'-only c c' FT

c-in-c'-only c c' (FVar x)

c-in-c'-only c c' (FNot FF)

c-in-c'-only c c' (FNot FT)

c-in-c'-only c c' (FNot (FVar x))

unfolding *c-in-c'-only-def* **by** *auto*

lemma *not-c-in-c'-symb-commute*:

not-c-in-c'-symb c c' ξ $\implies wf-conn\ c\ [\varphi, \psi] \implies \xi = conn\ c\ [\varphi, \psi]$

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

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

case (*not-c-in-c'-symb-r* $\varphi'\ \varphi''\ \psi'$) **note** *H = this*

hence $\psi = conn\ c'\ [\varphi'', \psi']$ **using** *conn-inj* **by** *auto*

have *wf-conn c [conn c' [φ'', ψ'], φ]*

using *H(1) wf-conn-no-arity-change length-Cons* **by** *metis*

thus *not-c-in-c'-symb c c' (conn c [ψ, φ])*

unfolding ψ **using** *not-c-in-c'-symb.intros(1)* *H* **by** *auto*

next

case (*not-c-in-c'-symb-l* $\varphi'\ \varphi''\ \psi'$) **note** *H = this*

hence $\varphi = conn\ c'\ [\varphi', \varphi'']$ **using** *conn-inj* **by** *auto*

moreover have *wf-conn c [ψ', conn c' [φ', φ'']]*

using *H(1) wf-conn-no-arity-change length-Cons* **by** *metis*

ultimately show *not-c-in-c'-symb c c' (conn c [ψ, φ])*

using *not-c-in-c'-symb.intros(2) conn-inj not-c-in-c'-symb-l.hyps*

not-c-in-c'-symb-l.prem(1,2) **by** *blast*

qed

lemma *not-c-in-c'-symb-commute'*:

wf-conn c [φ, ψ] $\implies c-in-c'-symb\ c\ c'\ (conn\ c\ [\varphi, \psi]) \longleftrightarrow c-in-c'-symb\ c\ c'\ (conn\ c\ [\psi, \varphi])$

using *not-c-in-c'-symb-commute wf-conn-no-arity-change* **by** (*metis length-Cons*)

lemma *not-c-in-c'-comm*:

assumes *wf*: *wf-conn c [φ, ψ]*

shows *c-in-c'-only c c' (conn c [φ, ψ])* \longleftrightarrow *c-in-c'-only c c' (conn c [ψ, φ])* (**is** $?A \longleftrightarrow ?B$)

proof –

have $?A \longleftrightarrow (c\text{-in-}c'\text{-symb } c \ c' \ (conn \ c \ [\varphi, \psi])$
 $\wedge (\forall \xi \in set \ [\varphi, \psi]. \ all\text{-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$
using *all-subformula-st-decomp wf unfolding c-in-c'-only-def by fastforce*
also have $\dots \longleftrightarrow (c\text{-in-}c'\text{-symb } c \ c' \ (conn \ c \ [\psi, \varphi])$
 $\wedge (\forall \xi \in set \ [\psi, \varphi]. \ all\text{-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$
using *not-c-in-c'-symb-commute' wf by auto*
also
have *wf-conn* $c \ [\psi, \varphi]$ **using** *wf-conn-no-arity-change wf by (metis length-Cons)*
hence $(c\text{-in-}c'\text{-symb } c \ c' \ (conn \ c \ [\psi, \varphi])$
 $\wedge (\forall \xi \in set \ [\psi, \varphi]. \ all\text{-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$
 $\longleftrightarrow ?B$
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 $\varphi1 \ \varphi2 \ \psi :: 'v \ propo$ **and** $x :: 'v$
shows
 $c\text{-in-}c'\text{-symb } c \ c' \ FT$
 $c\text{-in-}c'\text{-symb } c \ c' \ FF$
 $c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$
 $wf\text{-conn } c \ [conn \ c' \ [\varphi1, \varphi2], \psi] \implies wf\text{-conn } c' \ [\varphi1, \varphi2]$
 $\implies \neg c\text{-in-}c'\text{-only } c \ c' \ (conn \ c \ [conn \ c' \ [\varphi1, \varphi2], \psi])$
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 $c \ c' :: 'v \ connective$ **and** $\psi :: 'v \ propo$
shows $c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ \psi)$

proof –

{
fix $\xi :: 'v \ propo$
have $not\text{-}c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ \psi) \implies False$
apply (*induct FNot ψ 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 $\varphi :: 'v \ propo$
assumes $c: c = CAnd \vee c = COr$ **and** $c': c' = CAnd \vee c' = COr$
shows $\psi \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi \implies \exists \psi'. \ push\text{-conn-inside } c \ c' \ \psi \ \psi'$
apply (*induct ψ rule: propo-induct-arity*)
apply *auto[2]*

proof –

fix $\psi1 \ \psi2 \ \varphi': 'v \ propo$
assume $IH\psi1: \psi1 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi1 \implies Ex \ (push\text{-conn-inside } c \ c' \ \psi1)$
and $IH\psi2: \psi2 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi2 \implies Ex \ (push\text{-conn-inside } c \ c' \ \psi2)$
and $\varphi': \varphi' = FAnd \ \psi1 \ \psi2 \vee \varphi' = FOr \ \psi1 \ \psi2 \vee \varphi' = FImp \ \psi1 \ \psi2 \vee \varphi' = FEq \ \psi1 \ \psi2$
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 $\varphi': \varphi' = conn \ c \ [\psi1, \psi2]$

```

obtain  $a\ b$  where  $\psi1 = \text{conn } c' [a, b] \vee \psi2 = \text{conn } c' [a, b]$ 
  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  $\varphi': \varphi' \neq \text{conn } c\ [\psi1, \psi2]$ 
  have  $\forall \varphi\ c\ ca. \exists \varphi1\ \psi1\ \psi2\ \psi1'\ \psi2'\ \varphi2'. \text{conn } (c::'v\ \text{connective})\ [\varphi1, \text{conn } ca\ [\psi1, \psi2]] = \varphi$ 
     $\vee \text{conn } c\ [\text{conn } ca\ [\psi1', \psi2'], \varphi2'] = \varphi \vee c\text{-in-}c'\text{-symb } c\ ca\ \varphi$ 
  by (metis not-c-in-c'-symb.cases)
  hence  $Ex\ (\text{push-conn-inside } c\ c'\ \varphi')$ 
  by (metis (no-types) c\ c'\ n\ push-conn-inside-l\ push-conn-inside-r)
}
ultimately show  $Ex\ (\text{push-conn-inside } c\ c'\ \varphi')$  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-in-c'-symb-step-exists)
  }
  ultimately show ?thesis using noTB no-test-symb-step-exists[of c-in-c'-symb c c']
  unfolding c-in-c'-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 (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 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 no-T-F  $\varphi$ 

```

```

using wf no-T-F no-T-F-def subformula-into-subformula subformula-all-subformula-st
subformula-refl by (metis (no-types) in-set-conv-decomp)
hence  $\varphi'$ : no-T-F  $\varphi'$  using IH by blast

have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi')$ . no-T-F  $\zeta$  by (metis wf no-T-F no-T-F-def all-subformula-st-decomp)
hence  $n$ :  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi')$ . 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  $c$  ( $\xi @ \varphi' \# \xi'$ )
using wf wf-conn-no-arity-change by (metis wf-conn-no-arity-change-helper)
{
  fix  $x :: 'v$ 
  assume  $c = CT \vee c = CF \vee c = CVar\ x$ 
  hence False using wf by auto
  hence no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) by blast
}
moreover {
  assume  $c$ :  $c = CNot$ 
  hence  $\xi = [] \ \xi' = []$  using wf by auto
  hence no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ ))
    using  $c$  by (metis  $\varphi'$  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  $c$ :  $c \in \text{binary-connectives}$ 
  hence no-T-F-symb (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) using wf'  $n'$  no-T-F-symb.simps by fastforce
  hence no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) by (metis all-subformula-st-decomp-imp wf'  $n$  no-T-F-def)
}
ultimately show no-T-F (conn  $c$  ( $\xi @ \varphi' \# \xi'$ )) using connective-cases-arity by auto
qed

```

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

```

propo-rew-step (push-conn-inside  $c\ c'$ )  $\varphi\ \psi \implies \text{simple } \varphi \implies \text{simple } \psi$ 
apply (induct rule: propo-rew-step.induct)
apply (case-tac  $\varphi$ , 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  $\varphi\ \psi :: 'v$  propo
shows propo-rew-step (push-conn-inside  $c\ c'$ )  $\varphi\ \psi \implies \text{simple-not } \varphi \implies \text{simple-not } \psi$ 
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi\ \psi$ )
  thus ?case by (case-tac  $\varphi$ , auto simp add: push-conn-inside.simps)
next
  case (propo-rew-one-step-lift  $\varphi\ \varphi'\ ca\ \xi\ \xi'$ )
  thus ?case
    proof (case-tac  $ca$  rule: connective-cases-arity, auto)
      fix  $\varphi\ \varphi' :: 'v$  propo and  $c :: 'v$  connective and  $\xi\ \xi' :: 'v$  propo list
      assume rel: propo-rew-step (push-conn-inside  $c\ c'$ )  $\varphi\ \varphi'$ 
      assume simple  $\varphi$ 
      thus simple  $\varphi'$  using rel simple-propo-rew-step-push-conn-inside-inv by blast
    qed

```

```

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

  obtain  $a \ b$  where  $ab$ :  $\xi @ \varphi' \# \xi' = [a, b]$ 
  using wf  $ca$  list-length2-decomp wf-conn-bin-list-length
  by (metis (no-types) wf-conn-no-arity-change-helper)
  have  $\forall \zeta \in$  set  $(\xi @ \varphi \# \xi')$ . simple-not  $\zeta$ 
  by (metis wf all-subformula-st-decomp simple-not simple-not-def)
  hence  $\forall \zeta \in$  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 propo and  $\xi \xi'$  :: 'v propo list and  $c$  :: 'v connective
shows propo-rew-step (push-conn-inside  $c \ c'$ )  $\varphi \varphi' \implies$  wf-conn  $c (\xi @ \varphi \# \xi')$ 
 $\implies$  simple-not-symb (conn  $c (\xi @ \varphi \# \xi')$ )  $\implies$  simple-not-symb  $\varphi'$ 
 $\implies$  simple-not-symb (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 propo and  $ca$  :: 'v connective and  $\chi s \ \chi s'$  :: 'v propo list
assume simple-not-symb (conn  $c (\xi @$  conn  $ca (\chi s @ \varphi \# \chi s') \# \xi')$ )
and simple-not-symb (conn  $ca (\chi s @ \varphi' \# \chi s')$ )
and corr: wf-conn  $c (\xi @$  conn  $ca (\chi s @ \varphi \# \chi s') \# \xi')$ 
and  $c$ :  $c \in$  binary-connectives
have corr': wf-conn  $c (\xi @$  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  $\xi @$  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  $c (\xi @$  conn  $ca (\chi s @ \varphi' \# \chi s') \# \xi')$ )
  using  $c$  unfolding binary-connectives-def by auto

```

next

```

fix  $\varphi \varphi'$  :: 'v propo and  $ca$  :: 'v connective and  $\chi s \ \chi s'$  :: 'v 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  $\in$  binary-connectives
  obtain a b where ab:  $\chi s @ \varphi \# \chi s' = [a, b]$ 
    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*:
 $\text{push-conn-inside } c \ c' \ \varphi \ \psi \implies \psi \neq FT \wedge \psi \neq FF$
 by (induct rule: push-conn-inside.induct, auto)

lemma *push-conn-inside-inv*:
 fixes $\varphi \ \psi :: 'v \text{ propo}$
 assumes full (propo-rew-step (push-conn-inside c c') $\varphi \ \psi$)
 and no-equiv φ and no-imp φ and no-T-F-except-top-level φ and simple-not φ
 shows no-equiv ψ and no-imp ψ and no-T-F-except-top-level ψ and simple-not ψ

proof –

```

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

```

```

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

```

assms **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: push-conn-inside.induct, auto)
}
thus 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* :: $'v \text{ connective} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **for** $c :: 'v \text{ connective}$ **where**
simple-only-c-inside[*simp*]: $\text{simple } \varphi \implies \text{only-c-inside-symb } c \ \varphi \mid$
simple-cnot-only-c-inside[*simp*]: $\text{simple } \varphi \implies \text{only-c-inside-symb } c \ (FNot \ \varphi) \mid$
only-c-inside-into-only-c-inside: $\text{wf-conn } c \ l \implies \text{only-c-inside-symb } c \ (\text{conn } c \ l)$

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  $c :: 'v$  connective
  assumes  $c: c \neq CNot$ 
  shows only-c-inside-symb  $c$  ( $FNot\ \psi$ )  $\longleftrightarrow$  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-c-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-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-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-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
  {
    fix  $\varphi'$ 
    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-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-c-inside  $c\ \varphi \implies$  c-in-c'-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-conn  $ca\ l \implies ca \neq c \implies$  c-in-c'-symb  $c\ c'\ (conn\ ca\ l)$ 

```

proof –

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

qed

lemma $\text{only-c-inside-implies-c-in-c'-only}$:

assumes δ : $c \neq c'$ **and** c : $c = CAnd \vee c = COr$ **and** c' : $c' = CAnd \vee c' = COr$
shows $\text{only-c-inside } c \ \varphi \implies \text{c-in-c'-only } c \ c' \ \varphi$
unfolding c-in-c'-only-def $\text{all-subformula-st-def}$
using $\text{only-c-inside-implies-c-in-c'-symb}$
by ($\text{metis all-subformula-st-def assms(1) } c \ c' \ \text{only-c-inside-def subformula-trans}$)

lemma $\text{c-in-c'-symb-c-implies-only-c-inside}$:

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

using inv

proof ($\text{induct conn } c \ l$ arbitrary: l rule: $\text{propo-induct-arity}$)

case ($\text{nullary } x$)

thus ?case **by** ($\text{auto simp add: wf-conn-list assms}$)

next

case ($\text{unary } \varphi \ la$)

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

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

next

case ($\text{binary } \varphi1 \ \varphi2$)

note $\text{IH}\varphi1 = \text{this(1)}$ **and** $\text{IH}\varphi2 = \text{this(2)}$ **and** $\varphi = \text{this(3)}$ **and** $\text{only} = \text{this(5)}$ **and** $\text{wf} = \text{this(4)}$

and $\text{no-equiv} = \text{this(6)}$ **and** $\text{no-imp} = \text{this(7)}$ **and** $\text{simple-not} = \text{this(8)}$

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

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

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

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

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

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

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

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

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

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

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

proof

assume $c1c$: $c1 = c'$

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

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

```

have c-in-c'-only c c' (conn c [conn c' l1,  $\varphi 2$ ]) using c1c l only  $\varphi 1$  by auto
moreover have not-c-in-c'-symb c c' (conn c [conn c' l1,  $\varphi 2$ ])
  using l1  $\varphi 1$  c1c l local.wf not-c-in-c'-symb-l wf $\varphi 1$  by blast
ultimately show False using  $\varphi 1$  c1c l l1 local.wf not-c-in-c'-simp(4) wf $\varphi 1$  by blast
qed
hence ( $\varphi 1 = \text{conn } c \text{ l1} \wedge \text{wf-conn } c \text{ l1}$ )  $\vee$  ( $\exists \psi 1. \varphi 1 = \text{FNot } \psi 1$ )  $\vee$  simple  $\varphi 1$ 
  by (metis  $\varphi 1$  assms(1-3) c1-eq c1-imp simple.elims(3) wf $\varphi 1$  wf-conn-list(4) wf-conn-list(5-7))
moreover {
  assume  $\varphi 1 = \text{conn } c \text{ l1} \wedge \text{wf-conn } c \text{ l1}$ 
  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  $\exists \psi 1. \varphi 1 = \text{FNot } \psi 1$ 
  then obtain  $\psi 1$  where  $\varphi 1 = \text{FNot } \psi 1$  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: 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 [ $\varphi 1$ , 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 [ $\varphi 1$ , 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 ( $\varphi 2 = \text{conn } c \text{ l2} \wedge \text{wf-conn } c \text{ l2}$ )  $\vee$  ( $\exists \psi 2. \varphi 2 = \text{FNot } \psi 2$ )  $\vee$  simple  $\varphi 2$ 
  using c2-eq by (metis  $\varphi 2$  assms(1-3) c2-eq c2-imp simple.elims(3) wf $\varphi 2$  wf-conn-list(4-7))
moreover {
  assume  $\varphi 2 = \text{conn } c \text{ l2} \wedge \text{wf-conn } c \text{ l2}$ 
  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  $\exists \psi 2. \varphi 2 = \text{FNot } \psi 2$ 
  then obtain  $\psi 2$  where  $\varphi 2 = \text{FNot } \psi 2$  by metis
  hence only-c-inside c  $\varphi 2$ 

```

```

    by (metis all-subformula-st-def assms(1-3) connective.distinct(38,40) incφ2
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
  }
  moreover {
    assume simple φ2
    hence only-c-inside c φ2
    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-insideφ2: only-c-inside c φ2 by metis
  show ?case using l only-c-insideφ1 only-c-insideφ2 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 φ and no-imp φ and no-T-F-except-top-level φ and simple-not φ*
shows *no-equiv ψ and no-imp ψ and no-T-F-except-top-level ψ and simple-not ψ*
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 φ and
no-imp φ and
full (propo-rew-step pushConj) $\varphi \psi$ and
no-T-F-except-top-level φ and
simple-not φ
shows *and-in-or-only ψ*
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]:
 $\sim \text{or-in-and-only } (FOr \ (FAnd \ \psi1 \ \psi2) \ \varphi')$
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 *full (propo-rew-step pushDisj) $\varphi \ \psi$*
and *no-equiv φ and no-imp φ and no-T-F-except-top-level φ and simple-not φ*
shows *no-equiv ψ and no-imp ψ and no-T-F-except-top-level ψ and simple-not ψ*
using *push-conn-inside-inv assms unfolding pushDisj-def by metis+*

lemma *pushDisj-full-propo-rew-step*:
fixes $\varphi \ \psi :: 'v \text{ propo}$
assumes
no-equiv φ and
no-imp φ and
full (propo-rew-step pushDisj) $\varphi \ \psi$ and
no-T-F-except-top-level φ and
simple-not φ
shows *or-in-and-only ψ*
using *assms push-conn-inside-full-propo-rew-step*
unfolding *pushDisj-def or-in-and-only-def c-in-c'-only-def by (metis (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 *grouped-by* :: *'a connective \Rightarrow 'a propo \Rightarrow bool for c where*
simple-is-grouped[simp]: *simple $\varphi \Rightarrow$ grouped-by c φ |*
simple-not-is-grouped[simp]: *simple $\varphi \Rightarrow$ grouped-by c (FNot φ) |*
connected-is-group[simp]: *grouped-by c $\varphi \Rightarrow$ grouped-by c $\psi \Rightarrow$ wf-conn c $[\varphi, \psi]$*
 \Rightarrow *grouped-by c (conn c $[\varphi, \psi]$)*

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

lemma *only-c-inside-symb-c-eq-c'*:
only-c-inside-symb c (conn c' $[\varphi1, \varphi2]$) \Rightarrow $c' = CAnd \vee c' = COr \Rightarrow$ wf-conn c' $[\varphi1, \varphi2]$
 \Rightarrow *$c' = c$*
by (*induct conn c' $[\varphi1, \varphi2]$ rule: only-c-inside-symb.induct, auto simp add: conn-inj*)

lemma *only-c-inside-c-eq-c'*:

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

lemma *only-c-inside-imp-grouped-by*:

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

proof (*induct* φ *rule: propo-induct-arity*)

case (*nullary* φ x)

thus $?G \varphi$ **by** *auto*

next

case (*unary* ψ)

thus $?G (FNot \psi)$ **by** (*auto simp add: c*)

next

case (*binary* φ $\varphi 1$ $\varphi 2$)

note $IH\varphi 1 = this(1)$ **and** $IH\varphi 2 = this(2)$ **and** $\varphi = this(3)$ **and** $only = this(4)$

have $\varphi\text{-}conn$: $\varphi = conn\ c [\varphi 1, \varphi 2]$ **and** wf : $wf\text{-}conn\ c [\varphi 1, \varphi 2]$

proof –

obtain $c''\ l''$ **where** $\varphi\text{-}c'': \varphi = conn\ c''\ l''$ **and** wf : $wf\text{-}conn\ c''\ l''$

using *exists-c-conn* **by** *metis*

hence $l'': l'' = [\varphi 1, \varphi 2]$ **using** φ **by** (*metis wf-conn-list(4-7)*)

have *only-c-inside-symb* c (*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* $\varphi\ \varphi\text{-}c''\ conn\text{-}inj\ conn\text{-}inj\text{-}not(2)\ l''\ list.distinct(1)\ list.inject\ wf$

only-c-inside-symb.cases simple.simps(5-8))

thus $\varphi = conn\ c [\varphi 1, \varphi 2]$ **and** $wf\text{-}conn\ c [\varphi 1, \varphi 2]$ **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** $\varphi\text{-}conn\ connected\text{-}is\text{-}group\ local.wf$ **by** *blast*

qed

lemma *grouped-by-false*:

grouped-by c (*conn* c' [φ , ψ]) $\implies c \neq c' \implies wf\text{-}conn\ c' [\varphi, \psi] \implies False$

apply (*induct* *conn* c' [φ , ψ] *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* \implies '*a* *connective* \implies '*a* *propo* \implies *bool* **for** $c\ c'$ **where**

grouped-is-super-grouped[*simp*]: *grouped-by* $c\ \varphi \implies$ *super-grouped-by* $c\ c'\ \varphi$ |

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

\implies *super-grouped-by* $c\ c' (conn\ c' [\varphi, \psi])$

lemma *simple-cnf*[*simp*]:

super-grouped-by $c\ c'\ FT$

super-grouped-by $c\ c'\ FF$

```

super-grouped-by c c' (FVar x)
super-grouped-by c c' (FNot FT)
super-grouped-by c c' (FNot FF)
super-grouped-by c c' (FNot (FVar x))
by auto

lemma c-in-c'-only-super-grouped-by:
  assumes c: c = CAnd ∨ c = COr and c': c' = CAnd ∨ c' = COr and cc': c ≠ c'
  shows no-equiv φ ⇒ no-imp φ ⇒ simple-not φ ⇒ c-in-c'-only c c' φ
    ⇒ super-grouped-by c c' φ
    (is ?NE φ ⇒ ?NI φ ⇒ ?SN φ ⇒ ?C φ ⇒ ?S φ)
proof (induct φ rule: propo-induct-arity)
  case (nullary φ x)
  thus ?S φ by auto
next
  case (unary φ)
  hence simple-not-symb (FNot φ)
    using all-subformula-st-test-symb-true-phi unfolding simple-not-def by blast
  hence φ = FT ∨ φ = FF ∨ (∃ x. φ = FVar x) by (case-tac φ, auto)
  thus ?S (FNot φ) by auto
next
  case (binary φ φ1 φ2)
  note IHφ1 = this(1) and IHφ2 = this(2) and no-equiv = this(4) and no-imp = this(5)
    and simpleN = this(6) and c-in-c'-only = this(7) and φ' = this(3)
  {
    assume φ = FImp φ1 φ2 ∨ φ = FEq φ1 φ2
    hence False using no-equiv no-imp by auto
    hence ?S φ by auto
  }
  moreover {
    assume φ: φ = conn c' [φ1, φ2] ∧ wf-conn c' [φ1, φ2]
    have c-in-c'-only: c-in-c'-only c c' φ1 ∧ c-in-c'-only c c' φ2 ∧ c-in-c'-symb c c' φ
      using c-in-c'-only φ' unfolding c-in-c'-only-def by auto
    have super-grouped-by c c' φ1 using φ c' no-equiv no-imp simpleN IHφ1 c-in-c'-only by auto
    moreover have super-grouped-by c c' φ2
      using φ c' no-equiv no-imp simpleN IHφ2 c-in-c'-only by auto
    ultimately have ?S φ
      using super-grouped-by.intros(2) φ by (metis c wf-conn-helper-facts(5,6))
  }
  moreover {
    assume φ: φ = conn c [φ1, φ2] ∧ wf-conn c [φ1, φ2]
    hence only-c-inside c φ1 ∧ only-c-inside c φ2
      using c-in-c'-symb-c-implies-only-c-inside c c' c-in-c'-only 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 (conn c [φ1, φ2])
      unfolding only-c-inside-def using φ
      by (simp add: only-c-inside-into-only-c-inside all-subformula-st-decomp)
    hence grouped-by c φ using φ only-c-inside-imp-grouped-by c by blast
    hence ?S φ using super-grouped-by.intros(1) by metis
  }
  ultimately show ?S φ by (metis φ' c c' cc' 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* $\varphi \implies$ *no-imp* $\varphi \implies$ *simple-not* $\varphi \implies$ *or-in-and-only* $\varphi \implies$ *is-conj-with-TF* φ

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* φ == *is-conj-with-TF* $\varphi \wedge$ *no-T-F-except-top-level* φ

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* φ $\varphi' \implies$ *is-cnf* φ'

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

apply *auto*

proof –

fix φ φEq φImp φTB φNeg \varphiDisj :: '*v propo*

assume *Eq: full (propo-rew-step elim-equiv)* φ φEq

hence *no-equiv: no-equiv* φEq **using** *no-equiv-full-propo-rew-step-elim-equiv* **by** *blast*

assume *Imp: full (propo-rew-step elim-imp)* φEq φImp

hence *no-imp: no-imp* φImp **using** *no-imp-full-propo-rew-step-elim-imp* **by** *blast*

have *no-imp-inv: no-equiv* φImp **using** *no-equiv Imp elim-imp-inv* **by** *blast*

assume *TB: full (propo-rew-step elimTB)* φImp φTB

hence *noTB: no-T-F-except-top-level* φTB

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

have *noTB-inv: no-equiv* φTB *no-imp* φTB **using** *elimTB-inv TB no-imp no-imp-inv* **by** *blast+*

assume *Neg: full (propo-rew-step pushNeg)* φTB φNeg

hence *noNeg: simple-not* φNeg

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

have *noNeg-inv: no-equiv* φNeg *no-imp* φNeg *no-T-F-except-top-level* φNeg

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

assume *Disj: full (propo-rew-step pushDisj)* φNeg \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 φ Disj*
 using *or-in-and-only-conjunction-in-disj noDisj-inv no-Disj* by *blast*
 ultimately show *is-cnf φ Disj 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 *no-equiv $\varphi \implies$ no-imp $\varphi \implies$ simple-not $\varphi \implies$ and-in-or-only $\varphi \implies$ is-disj-with-TF φ*
 using *c-in-c'-only-super-grouped-by*
 unfolding *is-disj-with-TF-def and-in-or-only-def c-in-c'-only-def*
 by (*simp add: c-in-c'-only-def c-in-c'-only-super-grouped-by*)

definition *is-dnf* $:: 'a \text{ propo} \Rightarrow \text{bool}$ **where**

is-dnf $\varphi \longleftrightarrow$ is-disj-with-TF $\varphi \wedge$ no-T-F-except-top-level φ

9.3.1 Full DNF transform

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

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

lemma *dnf-rew-consistent: preserves-un-sat dnf-rew*

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 *dnf-transformation-correction*:

dnf-rew $\varphi \varphi' \implies$ is-dnf φ'
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 *elimTBFull* **where**

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

$ElimTBFull2[simp]: elimTBFull (FAnd \varphi FF) FF \mid$
 $ElimTBFull2'[simp]: elimTBFull (FAnd FF \varphi) FF \mid$

 $ElimTBFull3[simp]: elimTBFull (FOr \varphi FT) FT \mid$
 $ElimTBFull3'[simp]: elimTBFull (FOr FT \varphi) FT \mid$

 $ElimTBFull4[simp]: elimTBFull (FOr \varphi FF) \varphi \mid$
 $ElimTBFull4'[simp]: elimTBFull (FOr FF \varphi) \varphi \mid$

 $ElimTBFull5[simp]: elimTBFull (FNot FT) FF \mid$
 $ElimTBFull5'[simp]: elimTBFull (FNot FF) FT \mid$

 $ElimTBFull6-l[simp]: elimTBFull (FImp FT \varphi) \varphi \mid$
 $ElimTBFull6-l'[simp]: elimTBFull (FImp FF \varphi) FT \mid$
 $ElimTBFull6-r[simp]: elimTBFull (FImp \varphi FT) FT \mid$
 $ElimTBFull6-r'[simp]: elimTBFull (FImp \varphi FF) (FNot \varphi) \mid$

 $ElimTBFull7-l[simp]: elimTBFull (FEq FT \varphi) \varphi \mid$
 $ElimTBFull7-l'[simp]: elimTBFull (FEq FF \varphi) (FNot \varphi) \mid$
 $ElimTBFull7-r[simp]: elimTBFull (FEq \varphi FT) \varphi \mid$
 $ElimTBFull7-r'[simp]: elimTBFull (FEq \varphi FF) (FNot \varphi) \mid$

The transformation is still consistent.

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

proof –

```

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

```

Contrary to the theorem $\llbracket no\text{-equiv } ?\varphi; no\text{-imp } ?\varphi; ?\psi \preceq ?\varphi; \neg no\text{-T-F-symb-except-toplevel } ?\psi \rrbracket \implies \exists \psi'. elimTB \ ?\psi \ \psi'$, we do not need the assumption *no-equiv* φ and *no-imp* φ , since our transformation is more general.

lemma *no-T-F-symb-except-toplevel-step-exists'*:

```

fixes  $\varphi :: 'v \text{ propo}$ 
shows  $\psi \preceq \varphi \implies \neg no\text{-T-F-symb-except-toplevel } \psi \implies \exists \psi'. elimTBFull \ \psi \ \psi'$ 
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 (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 (elimTBFull (FNot  $\psi$ )) 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 (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 $\neg \text{no-}T\text{-}F\text{-except-top-level}$ φ and the existence of a rewriting step, still exists.

```

lemma no-T-F-except-top-level-rew':
  fixes  $\varphi :: 'v \text{ propo}$ 
  assumes noTB:  $\neg \text{no-}T\text{-}F\text{-except-top-level } \varphi$ 
  shows  $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elimTBFull } \psi \psi'$ 
proof –
  have test-symb-false-nullary:
     $\forall x. \text{no-}T\text{-}F\text{-symb-except-toplevel } (FF :: 'v \text{ propo}) \wedge \text{no-}T\text{-}F\text{-symb-except-toplevel } FT$ 
     $\wedge \text{no-}T\text{-}F\text{-symb-except-toplevel } (FVar (x :: 'v))$ 
  by auto
  moreover {
    fix  $c :: 'v \text{ connective}$  and  $l :: 'v \text{ propo list}$  and  $\psi :: 'v \text{ propo}$ 
    have  $H: \text{elimTBFull } (\text{conn } c \ l) \ \psi \implies \neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (\text{conn } c \ l)$ 
    by (case-tac ( $\text{conn } c \ l$ ) rule: elimTBFull.cases, simp-all)
  }
  ultimately show ?thesis
  using no-test-symb-step-exists[of no-T-F-symb-except-toplevel  $\varphi$  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  $\varphi \psi :: 'v \text{ propo}$ 
  assumes full (propo-rew-step elimTBFull)  $\varphi \psi$ 
  shows no-T-F-except-top-level  $\psi$ 
  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  $\varphi \psi \implies \text{no-}T\text{-}F \ \varphi \implies \text{no-}T\text{-}F \ \psi$ 
proof (induct rule: propo-rew-step.induct)
  fix  $\varphi' :: 'v \text{ propo}$  and  $\psi' :: 'v \text{ propo}$ 
  assume a1: no-T-F  $\varphi'$ 
  assume a2: elim-equiv  $\varphi' \psi'$ 
  have  $\forall x0 \ x1. (\neg \text{elim-equiv } (x1 :: 'v \text{ propo}) \ x0 \vee (\exists v2 \ v3 \ v4 \ v5 \ v6 \ v7. x1 = FEq \ v2 \ v3$ 
     $\wedge x0 = FAnd (FImp \ v4 \ v5) (FImp \ v6 \ v7) \wedge v2 = v4 \wedge v4 = v7 \wedge v3 = v5 \wedge v3 = v6))$ 
     $= (\neg \text{elim-equiv } x1 \ x0 \vee (\exists v2 \ v3 \ v4 \ v5 \ v6 \ v7. x1 = FEq \ v2 \ v3$ 
     $\wedge x0 = FAnd (FImp \ v4 \ v5) (FImp \ v6 \ v7) \wedge v2 = v4 \wedge v4 = v7 \wedge v3 = v5 \wedge v3 = v6))$ 
  by meson
  hence  $\forall p \ pa. \neg \text{elim-equiv } (p :: 'v \text{ propo}) \ pa \vee (\exists pb \ pc \ pd \ pe \ pf \ pg. p = FEq \ pb \ pc$ 
     $\wedge pa = FAnd (FImp \ pd \ pe) (FImp \ pf \ pg) \wedge pb = pd \wedge pd = pg \wedge pc = pe \wedge pc = pf)$ 
  using elim-equiv.cases by force
  thus no-T-F  $\psi'$  using a1 a2 by fastforce
next
  fix  $\varphi \varphi' :: 'v \text{ propo}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $c :: 'v \text{ connective}$ 
  assume rel: propo-rew-step elim-equiv  $\varphi \varphi'$ 
  and IH: no-T-F  $\varphi \implies \text{no-}T\text{-}F \ \varphi'$ 
  and corr: wf-conn  $c \ (\xi @ \varphi \ \# \ \xi')$ 
  and no-T-F: no-T-F ( $\text{conn } c \ (\xi @ \varphi \ \# \ \xi')$ )

```

```

{
  assume c: c = CNot
  hence empty:  $\xi = [] \ \xi' = []$  using corr by auto
  hence no-T-F  $\varphi$  using no-T-F c no-T-F-decomp-not by auto
  hence no-T-F (conn c ( $\xi @ \varphi' \# \xi'$ )) using c empty no-T-F-comp-not IH by auto
}
moreover {
  assume c: c  $\in$  binary-connectives
  obtain a b where ab:  $\xi @ \varphi \# \xi' = [a, b]$ 
    using corr c list-length2-decomp wf-conn-bin-list-length by metis
  hence  $\varphi$ :  $\varphi = a \vee \varphi = b$ 
    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  $l' \ ab$  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 φ
 shows no-T-F-except-top-level ψ

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

```

```

lemma propo-rew-step-ElimImp-no-T-F: propo-rew-step elim-imp  $\varphi \psi \implies$  no-T-F  $\varphi \implies$  no-T-F  $\psi$ 
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi' \psi'$ )
  thus no-T-F  $\psi'$ 
    using elim-imp.cases no-T-F-comp-not no-T-F-decomp(1,2)
    by (metis no-T-F-comp-expanded-explicit(2))
next
  case (propo-rew-one-step-lift  $\varphi \varphi' c \xi \xi'$ )
  note rel = this(1) and IH = this(2) and corr = this(3) and no-T-F = this(4)
  {

```



```

assume  $c$ :  $c = CNot$ 
hence  $empty$ :  $\xi = [] \ \xi' = []$  using  $corr$  by  $auto$ 
hence  $no-T-F \ \varphi$  using  $no-T-F \ c \ no-T-F-decomp-not$  by  $auto$ 
hence  $no-T-F \ (conn \ c \ (\xi @ \varphi' \# \xi'))$  using  $c \ empty \ no-T-F-comp-not \ IH$  by  $auto$ 
}
moreover {
  assume  $c$ :  $c \in binary-connectives$ 
  then obtain  $a \ b$  where  $ab$ :  $\xi @ \varphi \# \xi' = [a, b]$ 
    using  $corr \ list-length2-decomp \ wf-conn-bin-list-length$  by  $metis$ 
  hence  $\varphi$ :  $\varphi = a \vee \varphi = b$ 
    by ( $metis \ append-self-conv2 \ wf-conn-list-decomp(4) \ wf-conn-unary \ list.discI \ list.sel(3) \ nth-Cons-0 \ tl-append2$ )
  have  $\zeta$ :  $\forall \zeta \in set \ (\xi @ \varphi \# \xi'). \ no-T-F \ \zeta$  using  $ab \ c \ propo-rew-one-step-lift.prem$ s by  $auto$ 

  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  $\forall \zeta \in set \ (\xi @ \varphi' \# \xi'). \ no-T-F \ \zeta$  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  $H = this$ 
    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  $c :: 'v$  connective and  $\xi \xi' :: 'v$  propo list and  $\zeta \zeta' :: 'v$  propo
  assume rel: propo-rew-step elim-imp  $\zeta \zeta'$ 
  and incl:  $\zeta \preceq \varphi$ 
  and corr: wf-conn  $c (\xi @ \zeta \# \xi')$ 
  and no-T-F: no-T-F-symb-except-toplevel (conn  $c (\xi @ \zeta \# \xi')$ )
  and n: no-T-F-symb-except-toplevel  $\zeta'$ 
  have no-T-F-symb-except-toplevel (conn  $c (\xi @ \zeta' \# \xi')$ )
  proof
    have  $p$ : no-T-F-symb (conn  $c (\xi @ \zeta \# \xi')$ )
      by (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 corr wf-conn-no-T-F-symb-iff  $p$  by blast
    from rel incl have  $\zeta' \neq FT \wedge \zeta' \neq FF$ 
      apply (induction  $\zeta \zeta'$  rule: propo-rew-step.induct)
      apply (cases rule: elim-imp.cases, auto)
      using wf-conn-list(1,2) wf-conn-no-arity-change wf-conn-no-arity-change-helper
      by (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 auto
    moreover have  $c \neq CT \wedge c \neq CF$  using corr by auto
    ultimately show no-T-F-symb (conn  $c (\xi @ \zeta' \# \xi')$ )
      using corr wf-conn-no-arity-change no-T-F-symb-comp
      by (metis wf-conn-no-arity-change-helper)
  qed
}
ultimately show no-T-F-except-top-level  $\psi$ 
  using full-propo-rew-step-inv-stay-with-inc[of elim-imp no-T-F-symb-except-toplevel  $\varphi$ ]
  assms subformula-refl unfolding no-T-F-except-top-level-def by 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 $dnf\text{-rew}' :: 'a \text{ propo} \Rightarrow 'a \text{ propo} \Rightarrow \text{bool}$ **where** $dnf\text{-rew}' \equiv$
 (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 pushConj))

lemma *dnf-rew'-consistent: preserves-un-sat dnf-rew'*
by (*simp add: dnf-rew'-def elimEquiv-lifted-consistant elim-imp-lifted-consistant*
elimTBFull-consistent preserves-un-sat-OO pushConj-consistent pushNeg-lifted-consistant)

theorem *cnf-transformation-correction:*

dnf-rew' φ $\varphi' \implies$ is-dnf φ'

unfolding *dnf-rew'-def OO-def*

by (*meson and-in-or-only-conjunction-in-disj elimTBFull-full-propo-rew-step elim-equiv-inv'*
elim-imp-inv 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)))

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' φ $\varphi' \implies$ is-cnf φ'

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*

$I \models_l L \iff L \in I$

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

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

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

11.2.2 Atoms

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

lemma *atms-of-msmultiset[simp]*:
atms-of (*mset* a) = *atm-of* ' *set* a
by (*induct* a) *auto*

lemma *atms-of-ms-mset-unfold*:
atms-of-ms (*mset* ' b) = $(\bigcup_{x \in b. \text{atm-of } ' \text{set } x}$
unfolding *atms-of-ms-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-ms-empty-set[simp]*:
atms-of-ms $\{\}$ = $\{\}$
unfolding *atms-of-ms-def* **by** *auto*

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

lemma *atms-of-ms-mono*:
 $A \subseteq B \Rightarrow \text{atms-of-ms } A \subseteq \text{atms-of-ms } B$

unfolding *atms-of-ms-def* **by** *auto*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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*:
 $\text{atm-of } L' \in \text{atm-of } 'B \implies L' \in B \vee - L' \in B$
using *atms-of-s-def* **by** (*cases L' fastforce+*)

11.2.3 Totality

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

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

lemma *total-over-set-empty[simp]*:
 $\text{total-over-set } I \text{ } \{\}$
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-m-empty[simp]*:
 $\text{total-over-m } I \text{ } \{\}$
unfolding *total-over-m-def* **by** *auto*

lemma *total-over-set-single[iff]*:
 $\text{total-over-set } I \text{ } \{L\} \longleftrightarrow (\text{Pos } L \in I \vee \text{Neg } L \in I)$
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-set-insert[iff]*:
 $\text{total-over-set } I \text{ } (\text{insert } L \text{ } Ls) \longleftrightarrow ((\text{Pos } L \in I \vee \text{Neg } L \in I) \wedge \text{total-over-set } I \text{ } Ls)$
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-ms-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]:
 $total-over-m\ I\ (A \cup B) \longleftrightarrow (total-over-m\ I\ A \wedge total-over-m\ I\ B)$
unfolding *total-over-m-def* *total-over-set-def* **by** *auto*

lemma *total-over-m-insert*[iff]:
 $total-over-m\ I\ (insert\ a\ A) \longleftrightarrow (total-over-set\ I\ (atms-of\ a) \wedge total-over-m\ I\ A)$
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 $\exists I'. total-over-m\ (I \cup I')\ (A \cup B)$
 $\wedge (\forall x \in I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A)$
proof –
let $?I' = \{Pos\ v \mid v. v \in atms-of-ms\ B \wedge v \notin atms-of-ms\ A\}$
have $(\forall x \in ?I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A)$ **by** *auto*
moreover have $total-over-m\ (I \cup ?I')\ (A \cup B)$
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 $\exists I'. total-over-m\ (I \cup I')\ (A \cup B)$
 $\wedge (\forall x \in I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A) \wedge consistent-interp\ (I \cup I')$
proof –
let $?I' = \{Pos\ v \mid v. v \in atms-of-ms\ B \wedge v \notin atms-of-ms\ A \wedge Pos\ v \notin I \wedge Neg\ v \notin I\}$
have $(\forall x \in ?I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A)$ **by** *auto*
moreover have $total-over-m\ (I \cup ?I')\ (A \cup B)$
using *total* **unfolding** *total-over-m-def* *total-over-set-def* **by** *auto*
moreover have *consistent-interp* $(I \cup ?I')$
using *cons* **unfolding** *consistent-interp-def* **by** (*intro allI*) (*case-tac L*, *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** (*metis image-iff literal.exhaust-sel*)

lemma *total-over-set-literal-defined*:
assumes $\{\#A\# \} + D \in \psi s$

and *total-over-set* I (*atms-of-ms* ψ s)
shows $A \in I \vee \neg A \in I$
using *assms* **unfolding** *total-over-set-def* **by** (*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* ($I \cup \{L\}$) $\{\psi\}$
and $L: \neg L \in \# \psi \neg L \notin \# \psi$
shows *total-over-m* I $\{\psi\}$
unfolding *total-over-m-def* *total-over-set-def*

proof

fix l
assume $l: l \in \text{atms-of-ms } \{\psi\}$
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 $\text{atm-of } L \notin \text{atms-of-ms } \{\psi\}$
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 $L \in \# \psi \vee \neg L \in \# \psi$ **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* ($I \cup I'$) 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* ($I \cup I'$) ($\psi \cup \psi'$)
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*]: $I \models \text{replicate-mset } n \ L \longleftrightarrow n \neq 0 \wedge I \models_l L$
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-clss-true-clss*:
assumes $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$
and $\text{atms-of } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models L$
shows $I \models L$
using *assms* **unfolding** *true-clss-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 $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$
and $\text{atms-of-ms } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models_s L$
shows $I \models_s L$
using *assms* **unfolding** *true-clss-def true-lit-def Ball-def*
by (*meson atms-of-atms-of-ms-mono notin-vars-union-true-clss-true-clss subset-trans*)

11.2.5 Satisfiability

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

lemma *satisfiable-single*[simp]:
 $\text{satisfiable } \{\{\#L\#\}\}$
unfolding *satisfiable-def* **by** *fastforce*

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

lemma *satisfiable-decreasing*:
assumes $\text{satisfiable } (\psi \cup \psi')$
shows $\text{satisfiable } \psi$
using *assms* *total-over-m-union* **unfolding** *satisfiable-def* **by** *blast*

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

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
   $cons: consistent\text{-}interp\ I$  and
   $tot: total\text{-}over\text{-}m\ I\ CC$ 
  unfolding  $satisfiable\text{-}def$  by  $auto$ 
let  $?I = \{P. P \in I \wedge atm\text{-}of\ P \in atm\text{-}of\text{-}ms\ CC\}$ 

have  $I\text{-}CC: ?I \models_s CC$ 
  using  $I\text{-}CC$  unfolding  $true\text{-}clss\text{-}def\ Ball\text{-}def\ true\text{-}cls\text{-}def\ Bex\text{-}mset\text{-}def\ true\text{-}lit\text{-}def$ 
  by ( $smt\ atm\text{-}of\text{-}lit\text{-}in\text{-}atms\text{-}of\ atm\text{-}of\text{-}atms\text{-}of\text{-}ms\text{-}mono\ mem\text{-}Collect\text{-}eq\ subset\text{-}eq$ )

moreover have  $cons: consistent\text{-}interp\ ?I$ 
  using  $cons$  unfolding  $consistent\text{-}interp\text{-}def$  by  $auto$ 
moreover have  $total\text{-}over\text{-}m\ ?I\ CC$ 
  using  $tot$  unfolding  $total\text{-}over\text{-}m\text{-}def\ total\text{-}over\text{-}set\text{-}def$  by  $auto$ 
moreover
  have  $atms\text{-}CC\text{-}incl: atm\text{-}of\text{-}ms\ CC \subseteq atm\text{-}of\text{-}I$ 
    using  $tot$  unfolding  $total\text{-}over\text{-}m\text{-}def\ total\text{-}over\text{-}set\text{-}def\ atm\text{-}of\text{-}ms\text{-}def$ 
    by ( $auto\ simp\ add: atm\text{-}of\text{-}def\ atm\text{-}of\text{-}s\text{-}def[symmetric]$ )
  have  $atm\text{-}of\text{-} ?I = atm\text{-}of\text{-}ms\ CC$ 
    using  $atms\text{-}CC\text{-}incl$  unfolding  $atms\text{-}of\text{-}ms\text{-}def$  by  $force$ 
ultimately show  $?B$  by  $auto$ 
qed

```

11.2.6 Entailment for Multisets of Clauses

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

lemma $true\text{-}cls\text{-}mset\text{-}empty[simp]: I \models_m \{\#\}$
unfolding $true\text{-}cls\text{-}mset\text{-}def$ **by** $auto$

lemma $true\text{-}cls\text{-}mset\text{-}singleton[iff]: I \models_m \{\#C\# \} \longleftrightarrow I \models C$
unfolding $true\text{-}cls\text{-}mset\text{-}def$ **by** ($auto\ split: split\text{-}if\text{-}asm$)

lemma $true\text{-}cls\text{-}mset\text{-}union[iff]: I \models_m CC + DD \longleftrightarrow I \models_m CC \wedge I \models_m DD$
unfolding $true\text{-}cls\text{-}mset\text{-}def$ **by** $fastforce$

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

lemma $true\text{-}cls\text{-}mset\text{-}mono: set\text{-}mset\ DD \subseteq set\text{-}mset\ CC \Longrightarrow I \models_m CC \Longrightarrow I \models_m DD$
unfolding $true\text{-}cls\text{-}mset\text{-}def\ subset\text{-}iff$ **by** $auto$

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

lemma $true\text{-}cls\text{-}mset\text{-}increasing\text{-}r[simp]:$
 $I \models_m CC \Longrightarrow I \cup J \models_m CC$
unfolding $true\text{-}cls\text{-}mset\text{-}def$ **by** $auto$

theorem $true\text{-}cls\text{-}remove\text{-}unused:$
assumes $I \models \psi$
shows $\{v \in I. atm\text{-}of\ v \in atm\text{-}of\ \psi\} \models \psi$
using $assms$ **unfolding** $true\text{-}cls\text{-}def\ atm\text{-}of\text{-}def$ **by** $auto$

theorem *true-clss-remove-unused*:
assumes $I \models_s \psi$
shows $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \models_s \psi$
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-clss-remove-unused[of I]*)
moreover have $\{v \in I. \text{atm-of } v \in \text{atms-of } x\} \subseteq \{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\}$
using $\langle x \in \psi \rangle$ **by** (*auto simp add: atms-of-ms-def*)
ultimately show $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \models x$
using *true-clss-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-clss-remove-unused II'* **by** *blast*
moreover have $?I \subseteq I$ **using** H **by** *auto*
ultimately show *?thesis* **using** *true-clss-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-ms-empty*:
fixes $\psi :: 'v \text{ clauses}$
assumes $\text{atms-of-ms } \psi = \{\}$
shows $\psi = \{\} \vee \psi = \{\{\#\}\}$
using *assms* **by** (*auto simp add: atms-of-ms-def*)

lemma *consistent-interp-disjoint*:
assumes *consI: consistent-interp I*
and *disj: atms-of-s A \cap atms-of-s I = $\{\}$*
and *consA: consistent-interp A*
shows *consistent-interp (A \cup I)*
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 *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 *total-over-m* $I \ \psi$
shows *total-over-m* $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \ \psi$
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 *insert* $a \ M' \models D$
and *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 *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 *Pos* $p \in \# A$ **and** *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* ψ
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*:

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*:

tautology ($\{\#a\# \} + L$) \longleftrightarrow *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* χ

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 $I \models \varphi \vee$ *tautology* φ
using *assms* **unfolding** *true-cls-def* **by** *auto*

lemma *tautology-imp-tautology*:

fixes $\chi \chi' :: 'v$ *clause*
assumes $\forall I. total-over-m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$ **and** *tautology* χ
shows *tautology* χ' **unfolding** *tautology-def*

proof (*intro allI HOL.impI*)

fix $I :: 'v$ *literal set*
assume *totI*: *total-over-set* I (*atms-of* χ')
let $?I' = \{Pos\ v \mid v. v \in atms-of\ \chi \wedge v \notin atms-of-s\ I\}$
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 $\bigwedge L. L \in\# \chi' \Longrightarrow L \notin ?I'$
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* :: $'a$ *clause* \Rightarrow $'a$ *clause* \Rightarrow *bool* (**infix** \models_f 49) **where**

$\psi \models_f \chi \longleftrightarrow (\forall I. total-over-m\ I\ (\{\psi\} \cup \{\chi\}) \longrightarrow consistent-interp\ I \longrightarrow I \models \psi \longrightarrow I \models \chi)$

definition *true-cls-clss* :: $'a$ *clause* \Rightarrow $'a$ *clauses* \Rightarrow *bool* (**infix** \models_{fs} 49) **where**

$\psi \models_{fs} \chi \longleftrightarrow (\forall I. total-over-m\ I\ (\{\psi\} \cup \chi) \longrightarrow consistent-interp\ I \longrightarrow I \models \psi \longrightarrow I \models_s \chi)$

definition *true-clss-cls* :: $'a$ *clauses* \Rightarrow $'a$ *clause* \Rightarrow *bool* (**infix** \models_p 49) **where**

$N \models_p \chi \longleftrightarrow (\forall I. total-over-m\ I\ (N \cup \{\chi\}) \longrightarrow consistent-interp\ I \longrightarrow I \models_s N \longrightarrow I \models \chi)$

definition *true-clss-clss* :: $'a$ *clauses* \Rightarrow $'a$ *clauses* \Rightarrow *bool* (**infix** \models_{ps} 49) **where**

$N \models_{ps} N' \longleftrightarrow (\forall I. total-over-m\ I\ (N \cup N') \longrightarrow consistent-interp\ I \longrightarrow I \models_s N \longrightarrow I \models_s N')$

lemma *true-cls-cls-refl[simp]*:

$A \models_f A$
unfolding *true-cls-cls-def* **by** *auto*

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

$a \models_f C \Longrightarrow insert\ a\ A \models_p C$
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-cls-def* **by** *auto*

lemma *true-clss-clss-true-cls-clss*[iff]:
 $\{\chi\} \models_{ps} \psi \iff \chi \models_{fs} \psi$
unfolding *true-clss-clss-def* *true-cls-clss-def* **by** *auto*

lemma *true-clss-clss-empty*[simp]:
 $N \models_{ps} \{\}$
unfolding *true-clss-clss-def* **by** *auto*

lemma *true-clss-cls-subset*:
 $A \subseteq B \implies A \models_p CC \implies B \models_p CC$
unfolding *true-clss-cls-def* *total-over-m-union* **by** (*simp add: total-over-m-subset true-clss-mono*)

lemma *true-clss-cs-mono-l*[simp]:
 $A \models_p CC \implies A \cup B \models_p CC$
by (*auto intro: true-clss-cls-subset*)

lemma *true-clss-cs-mono-l2*[simp]:
 $B \models_p CC \implies A \cup B \models_p CC$
by (*auto intro: true-clss-cls-subset*)

lemma *true-clss-cls-mono-r*[simp]:
 $A \models_p CC \implies A \models_p CC + CC'$
unfolding *true-clss-cls-def* *total-over-m-union* *total-over-m-sum* **by** *blast*

lemma *true-clss-cls-mono-r'*[simp]:
 $A \models_p CC' \implies A \models_p CC + CC'$
unfolding *true-clss-cls-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-cls-in*[simp]:
 $CC \in A \implies A \models_p CC$
unfolding *true-clss-cls-def* *true-clss-def* *total-over-m-union* **by** *fastforce*

lemma *true-clss-cls-insert-l*[simp]:
 $A \models_p C \implies \text{insert } a \ A \models_p C$
unfolding *true-clss-cls-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]*:

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

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:  $\forall x \in I'. \text{atm-of } x \in \text{atms-of-ms } D \wedge \text{atm-of } x \notin \text{atms-of-ms } (A \cup C)$ 
      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]*:

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

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* \langle *consistent-interp* $I \rangle$
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* \langle *consistent-interp* $I \rangle$ **unfolding** *true-clss-clss-def* **by** *auto*
ultimately have $I \models D + C$ **using** \langle *consistent-interp* $I \rangle$ *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  $L \langle \text{consistent-interp } I \rangle$  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  $\langle I \models_s N \rangle$  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  $L \notin \# D \wedge -L \notin \# D$ 
    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 $(A \# \cup B) = \text{atms-of } A \cup \text{atms-of } B$

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 $L: L \in I \vee -L \in I$

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* $\langle \text{consistent-interp } I \rangle$

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* $\langle \text{consistent-interp } I \rangle$ **unfolding** *true-clss-cls-def* **by** *auto*

ultimately **have** $I \models D \# \cup C$ **using** $\langle \text{consistent-interp } I \rangle$ **unfolding** *consistent-interp-def* **by** *auto*

}

moreover {

assume $L: L \notin I \wedge -L \notin I$

let $?I' = I \cup \{L\}$

have *consistent-interp* $?I'$ **using** $L \langle \text{consistent-interp } I \rangle$ **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** $\langle I \models_s N \rangle$ **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 $L \notin \# D \wedge -L \notin \# D$
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]*:

$(\exists I. \text{consistent-interp } I \wedge I \models_s \varphi) \longleftrightarrow \text{satisfiable } \varphi$ (**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 $?I' = \{Pos\ v \mid v. v \notin \text{atms-of-s } I \wedge v \in \text{atms-of-ms } \varphi\}$

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* $I \implies I \models_s \varphi \implies \text{satisfiable } \varphi$

using *satisfiable-carac* **by** *metis*

11.3 Subsumptions

lemma *subsumption-total-over-m*:

assumes $A \subseteq \# B$

shows *total-over-m* $I \{B\} \implies \text{total-over-m } I \{A\}$

using *assms 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 $(D - \text{replicate-mset } (\text{count } D\ L)\ L - \text{replicate-mset } (\text{count } D\ (-L))\ (-L))$

$= \text{atms-of } D - \{\text{atm-of } L\}$

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 $(\forall I. \text{total-over-}m\ I\ \{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \vee \text{tautology}\ \varphi$
 using *assms*
proof (*induct card* $\{Pos\ v \mid v. v \in \text{atms-of}\ D \wedge v \notin \text{atms-of}\ C\}$ *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 $\text{atms-of}\ D \subseteq \text{atms-of}\ C$ **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** $IH = \text{this}(1)$ **and** $\text{card} = \text{this}(2)$ **and** $H = \text{this}(3)$ **and** $\text{incl} = \text{this}(4)$
 let $?atms = \{Pos\ v \mid v. v \in \text{atms-of}\ D \wedge v \notin \text{atms-of}\ C\}$
 have *finite* $?atms$ **by** *auto*
 then obtain L where $L: L \in ?atms$
 using *card* **by** (*metis* (*no-types*, *lifting*) *Collect-empty-eq card-0-eq mem-Collect-eq*
nat.simps(3))
 let $?D' = D - \text{replicate-mset}\ (\text{count}\ D\ L)\ L - \text{replicate-mset}\ (\text{count}\ D\ (-L))\ (-L)$
 have $\text{atms-of-}D: \text{atms-of-}ms\ \{D\} \subseteq \text{atms-of-}ms\ \{?D'\} \cup \{\text{atm-of}\ L\}$ **by** *auto*

 {
 fix I
 assume $\text{total-over-}m\ I\ \{?D'\}$
 then have *tot*: $\text{total-over-}m\ (I \cup \{L\})\ \{D\}$
 unfolding *total-over-}m-def total-over-set-def* **using** *atms-of-D* **by** *auto*

 assume *IDL*: $I \models ?D'$
 then have $I \cup \{L\} \models D$ **unfolding** *true-cls-def* **by** *force*
 then have $I \cup \{L\} \models \varphi$ **using** $H\ tot$ **by** *auto*

 moreover
 have *tot'*: $\text{total-over-}m\ (I \cup \{-L\})\ \{D\}$
 using *tot* **unfolding** *total-over-}m-def total-over-set-def* **by** *auto*
 have $I \cup \{-L\} \models D$ **using** *IDL* **unfolding** *true-cls-def* **by** *force*
 then have $I \cup \{-L\} \models \varphi$ **using** $H\ tot'$ **by** *auto*
 ultimately have $I \models \varphi \vee \text{tautology}\ \varphi$
 using $L\ \text{remove-literal-in-model-tautology}$ **by** *force*
 } **note** $H' = \text{this}$

 have $L \notin\# C$ **and** $-L \notin\# C$ **using** $L\ \text{atm-iff-pos-or-neg-lit}$ **by** *force+*
 then have $C\text{-in-}D': C \subseteq\# ?D'$ **using** $\langle C \subseteq\# D \rangle$ **by** (*auto simp add: subseteq-mset-def*)
 have $\text{card}\ \{Pos\ v \mid v. v \in \text{atms-of}\ ?D' \wedge v \notin \text{atms-of}\ C\} <$
 $\text{card}\ \{Pos\ v \mid v. v \in \text{atms-of}\ D \wedge v \notin \text{atms-of}\ C\}$
 using L **by** (*auto intro!*: *psubset-card-mono*)
 then show *?case*
 using $IH\ C\text{-in-}D'\ H'$ **unfolding** *card[symmetric]* **by** *blast*
qed

11.4 Removing Duplicates

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

tautology (*remdups-mset* C) \longleftrightarrow *tautology* C

unfolding *tautology-decomp* **by** *auto*

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

unfolding *atms-of-def* **by** *auto*

lemma *true-cls-remdups-mset*[*iff*]: $I \models \text{remdups-mset } C \longleftrightarrow I \models C$
unfolding *true-cls-def* **by** *auto*

lemma *true-clss-cls-remdups-mset*[*iff*]: $A \models_p \text{remdups-mset } C \longleftrightarrow A \models_p C$
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 $\neg \text{finite } \text{vars} \vee \text{vars} = \{\}$
 then $\{\{\#\}\}$
 else
 let *cls'* = *build-all-simple-clss* (*vars* - {*Min vars*}) in
 $\{\{\#Pos (\text{Min } \text{vars})\# \} + \chi \mid \chi. \chi \in \text{cls}'\} \cup$
 $\{\{\#Neg (\text{Min } \text{vars})\# \} + \chi \mid \chi. \chi \in \text{cls}'\} \cup$
 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*]:
 $\neg \text{finite } \text{vars} \vee \text{vars} = \{\} \implies \text{build-all-simple-clss } \text{vars} = \{\{\#\}\}$
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
 $\text{finite } \text{vars} \wedge \text{vars} \neq \{\} \implies \text{build-all-simple-clss } (\text{vars}::'\text{v}::\text{linorder set}) =$
 $\{\{\#Pos (\text{Min } \text{vars})\# \} + \chi \mid \chi. \chi \in \text{cls}\}$
 $\cup \{\{\#Neg (\text{Min } \text{vars})\# \} + \chi \mid \chi. \chi \in \text{cls}\}$
 $\cup \text{cls}$
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* = $\{\}$ $\vee \neg \text{finite } \text{atms}$
 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 ∈ build-all-simple-clss atms and
  finite atms
shows atms-of x ⊆ atms ∧ ¬tautology x ∧ distinct-mset x
using assms
proof (induct card atms arbitrary: atms x)
  case (0 atms)
  then show ?case by auto
next
  case (Suc n) note IH = this(1) and card = this(2) and x = this(3) and finite = this(4)
  obtain v where v ∈ atms and v: v = Min atms
  using Min-in card local.finite by fastforce

  let ?atms' = atms - {v}
  have build-all-simple-clss atms
    = {{#Pos v#} + χ |χ. χ ∈ build-all-simple-clss (?atms')}
      ∪ {{#Neg v#} + χ |χ. χ ∈ build-all-simple-clss (?atms')}
      ∪ build-all-simple-clss (?atms')
  using build-all-simple-clss-simps-else[of atms] finite ⟨v ∈ atms⟩ unfolding v
  by (metis emptyE)
  then consider
    (Pos) χ φ where x = {#φ#} + χ and χ ∈ build-all-simple-clss (?atms') and
    φ = Pos v ∨ φ = Neg v
  | (In) x ∈ build-all-simple-clss (?atms')
  using x by auto
  then show ?case
  proof cases
    case In
    then show ?thesis using card finite IH[of ?atms'] ⟨v ∈ atms⟩ by fastforce
  next
    case Pos note x-χ = this(1) and χ = this(2) and φ = this(3)
    have
      atms-of χ ⊆ atms - {v} and
      ¬ tautology χ and
      distinct-mset χ
    using card finite IH[of ?atms' χ] ⟨v ∈ atms⟩ x-χ χ by auto
    moreover then have count χ (Neg v) = 0
    using ⟨v ∈ atms⟩ unfolding x-χ by (metis Diff-insert-absorb Set.set-insert
      atm-iff-pos-or-neg-lit grOI subset-iff)
    moreover have count χ (Pos v) = 0
    using ⟨atms-of χ ⊆ atms - {v}⟩ by (meson Diff-iff atm-iff-pos-or-neg-lit
      contra-subsetD insertI1 not-gr0)
    ultimately show ?thesis
    using ⟨v ∈ atms⟩ φ unfolding x-χ
    by (auto simp add: tautology-add-single distinct-mset-add-single)
  qed
qed

```

lemma *cls-in-build-all-simple-clss*:

shows {#} ∈ 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 atms :: 'v :: linorder set
  assumes finite atms
  shows card (build-all-simple-clss atms)  $\leq 3^{\wedge}(\text{card atms})$ 
  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)  $\leq 3^{\wedge}(\text{card atms})$  by auto
  }
  moreover {
    let ?P = {{#Pos (Min atms)#} +  $\chi$  |  $\chi. \chi \in \text{build-all-simple-clss (atms - \{Min atms\})}$ }
    let ?N = {{#Neg (Min atms)#} +  $\chi$  |  $\chi. \chi \in \text{build-all-simple-clss (atms - \{Min atms\})}$ }
    let ?Z = build-all-simple-clss (atms - {Min atms})
    assume atms: atms  $\neq \{\}$ 
    then have min: Min atms  $\in$  atms using Min-in finite by auto
    then have card-atms-1: card atms  $\geq 1$  by (simp add: Suc-leI atms card-gt-0-iff local.finite)
    have card (build-all-simple-clss atms) = card (?P  $\cup$  ?N  $\cup$  ?Z) using atms finite by simp
    moreover
      have  $\bigwedge M \text{ Ma. card } ((M::'v \text{ literal multiset set}) \cup \text{Ma}) \leq \text{card Ma} + \text{card M}$ 
        by (simp add: add commute card-Un-le)
      then have card (?P  $\cup$  ?N  $\cup$  ?Z)  $\leq$  card ?Z + (card ?P + card ?N)
        by (meson Nat.le-trans card-Un-le nat-add-left-cancel-le)
      then have card (?P  $\cup$  ?N  $\cup$  ?Z)  $\leq$  card ?P + card ?N + card ?Z

      by presburger
    also
      have PZ: card ?P  $\leq$  card ?Z
        by (simp add: Setcompr-eq-image build-all-simple-clss-finite card-image-le)
      have NZ: card ?N  $\leq$  card ?Z
        by (simp add: Setcompr-eq-image build-all-simple-clss-finite card-image-le)
      have card ?P + card ?N + card ?Z  $\leq$  card ?Z + card ?Z + card ?Z
        using PZ NZ by linarith
      finally have card (build-all-simple-clss atms)  $\leq$  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  $\geq 1$ ) min by auto
      then have card ?Z  $\leq 3^{\wedge}(\text{card atms} - 1)$  using IH finite' card by metis
    moreover
      have (3::nat)  $\wedge$  (card atms - 1) + 3  $\wedge$  (card atms - 1) + 3  $\wedge$  (card atms - 1)
        = 3 * 3  $\wedge$  (card atms - 1) by simp
      then have (3::nat)  $\wedge$  (card atms - 1) + 3  $\wedge$  (card atms - 1) + 3  $\wedge$  (card atms - 1)
        = 3  $\wedge$  (card atms) by (metis card card-Suc-Diff1 local.finite min power-Suc)
      ultimately have card (build-all-simple-clss atms)  $\leq 3^{\wedge}(\text{card atms})$  by linarith
    }
  ultimately show card (build-all-simple-clss atms)  $\leq 3^{\wedge}(\text{card atms})$  by metis
qed

```

lemma *build-all-simple-clss-mono-disj*:

assumes $atms \cap atms' = \{\}$ **and** *finite* $atms$ **and** *finite* $atms'$

shows $build-all-simple-clss\ atms \subseteq build-all-simple-clss\ (atms \cup atms')$

using *assms*

proof (*induct card (atms \cup 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 $?min = Min\ (atms \cup atms')$

have $m: ?min \in atms \vee ?min \in atms'$ **by** (*metis Min-in Un-iff c card-eq-0-iff nat.distinct(1)*)

moreover {

assume $min: ?min \in atms'$

then have $min': ?min \notin atms$ **using** *disj* **by** *auto*

then have $atms = atms - \{?min\}$ **by** *fastforce*

then have $n = card\ (atms \cup (atms' - \{?min\}))$

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 $atms \cap (atms' - \{?min\}) = \{\}$ **using** *disj* **by** *auto*

moreover have $finite\ (atms' - \{?min\})$ **using** $finite'$ **by** *auto*

ultimately have $build-all-simple-clss\ atms \subseteq build-all-simple-clss\ (atms \cup (atms' - \{?min\}))$

using $IH[of\ atms\ atms' - \{?min\}]\ finite$ **by** *metis*

moreover have $atms \cup (atms' - \{?min\}) = (atms \cup atms') - \{?min\}$ **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 $min: ?min \in atms$

moreover have $min': ?min \notin atms'$ **using** *disj min* **by** *auto*

moreover have $atms' - \{?min\} = atms'$

using $\langle ?min \notin atms' \rangle$ **by** *fastforce*

ultimately have $n = card\ (atms - \{?min\} \cup atms')$

by (*metis Min-in Un-Diff c card-0-eq card-Diff-singleton-if diff-Suc-1 finite' finite-UnI finite nat.distinct(1)*)

moreover have $finite\ (atms - \{?min\})$ **using** $finite$ **by** *auto*

moreover have $(atms - \{?min\}) \cap atms' = \{\}$ **using** *disj* **by** *auto*

ultimately have $build-all-simple-clss\ (atms - \{?min\})$

$\subseteq build-all-simple-clss\ ((atms - \{?min\}) \cup atms')$

using $IH[of\ atms - \{?min\}\ atms']\ finite'$ **by** *metis*

moreover have $build-all-simple-clss\ atms$

$= \{\{\#Pos\ (Min\ atms)\#\} + \chi \mid \chi. \chi \in build-all-simple-clss\ (?atms')\}$

$\cup \{\{\#Neg\ (Min\ atms)\#\} + \chi \mid \chi. \chi \in build-all-simple-clss\ (?atms')\}$

$\cup build-all-simple-clss\ (?atms')$

using $build-all-simple-clss-simps-else[of\ atms]\ finite\ min$ **by** (*metis emptyE*)

moreover

let $?mcls = build-all-simple-clss\ (atms \cup atms' - \{?min\})$

have $build-all-simple-clss\ (atms \cup atms')$

$= \{\{\#Pos\ (?min)\#\} + \chi \mid \chi. \chi \in ?mcls\} \cup \{\{\#Neg\ (?min)\#\} + \chi \mid \chi. \chi \in ?mcls\} \cup ?mcls$

using $build-all-simple-clss-simps-else[of\ atms \cup atms']\ finite'\ min$

by (*metis c card-eq-0-iff nat.distinct(1)*)

moreover have $atms \cup atms' - \{?min\} = atms - \{?min\} \cup atms'$

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  $\subseteq$  atms'
  shows build-all-simple-clss atms  $\subseteq$  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 χ) arbitrary: χ*)

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 χ : *L* $\in\#$ χ and

L-min: *atm-of* *L* = *Min* (*atms-of* χ) and

mL χ : $\neg \neg L \in\#$ χ

by (metis *Min-in c card-0-eq literal.sel*(1,2) *nat.distinct*(1) *tautology-minus*)

then have χL : $\chi = (\chi - \{\#L\}) + \{\#L\}$ by auto

have *atm χ* : *atms-of* χ = *atms-of* ($\chi - \{\#L\}$) \cup {*atm-of* *L*}

using *arg-cong*[OF χL , of *atms-of*] by simp

have *a χ* : *atms-of* ($\chi - \{\#L\}$) = (*atms-of* χ) – {*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* $\in\#$ $\chi - \{\#L\} \vee \neg L \in\#$ $\chi - \{\#L\}$

using *l' l* by (auto simp add: *atm-of-eq-atm-of*)

moreover have *L* $\notin\#$ $\chi - \{\#L\}$ using $\langle L \in\# \chi \rangle$ simp unfolding *distinct-mset-def* by auto

ultimately have *False* using *mL χ* by auto

}

ultimately show *v* \in *atms-of* $\chi - \{\#L\}$

by (auto dest: *atm-of-lit-in-atms-of split: split-if-asm*)

next

show *atms-of* $\chi - \{\#L\} \subseteq$ *atms-of* ($\chi - \{\#L\}$) using *atm χ* 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-clss.simps[of atms-of  $\chi$ ] Let-def)
qed

lemma simplified-in-build-all:
  assumes finite  $\psi$  and distinct-mset-set  $\psi$  and  $\forall \chi \in \psi. \neg$ tautology  $\chi$ 
  shows  $\psi \subseteq$  build-all-simple-clss (atms-of-ms  $\psi$ )
  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  $\neg$ tautology  $\chi$ 
    using simp tauto unfolding distinct-mset-set-def by auto
  from distinct-mset-not-tautology-implies-in-build-all-simple-clss[OF this]
  have  $\chi$ :  $\chi \in$  build-all-simple-clss (atms-of  $\chi$ ) .
  then have  $\psi \subseteq$  build-all-simple-clss (atms-of-ms  $\psi$ ) using IH simp tauto by auto
  moreover
    have atms-of-ms  $\psi \subseteq$  atms-of-ms (insert  $\chi \psi$ ) unfolding atms-of-ms-def atms-of-def by force
  ultimately
    have  $\psi \subseteq$  build-all-simple-clss (atms-of-ms (insert  $\chi \psi$ ))
      by (meson atms-of-ms-finite build-all-simple-clss-mono dual-order.trans finite.insertI local.finite)
  moreover
    have  $\chi \in$  build-all-simple-clss (atms-of-ms (insert  $\chi \psi$ ))
      using  $\chi$  finite build-all-simple-clss-mono[of atms-of-ms (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]:  $I \neq \{\}$   $\Longrightarrow$  insert L I  $\models_e x \longleftrightarrow \{L\} \models_e x \vee I \models_e x$ 
  assumes entail-union[simp]:  $I \models_e A \Longrightarrow I \cup I' \models_e A$ 
begin

```

```

definition entails :: 'a set  $\Rightarrow$  'b set  $\Rightarrow$  bool (infix  $\models_{es}$  50) where
  I  $\models_{es} A \longleftrightarrow (\forall a \in A. I \models_e a)$ 

```

```

lemma entails-empty[simp]:
  I  $\models_{es} \{\}$ 
  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]:  $I \models_{es} \text{insert } C \ DD \longleftrightarrow I \models_e C \wedge I \models_{es} DD$ 
  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]:  $I \models_{es} N \implies I \models_{es} \text{Set.remove } a \ N$ 
  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

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

```

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 =  $J \cup \{Pos (atm-of P) | P. P \in \# C \wedge atm-of P \notin atm-of 'J\}$ 
have  $I \subseteq ?J$  using  $\langle I \subseteq J \rangle$  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-ms-def
  apply (auto simp add: atms-of-def)
  apply (case-tac a  $a \in N - \{C\}$ )
  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  $\{v \in ?J. atm-of v \in atms-of-ms (N - \{C\})\} \subseteq J$ 
  using tot unfolding total-over-m-def total-over-set-def
  by (auto intro!: rev-image-eqI)
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  $I \models_{sxt} A$ 
  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  $\neg consistent-interp I$ 
  shows  $I \not\models_{sxt} A$ 
  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* :: $'v \text{ clauses} \Rightarrow 'v \text{ clauses} \Rightarrow \text{bool}$ **for** $N :: 'v \text{ clause set}$ **where**

tautology-deletion:

$(A + \{\#Pos P\} + \{\#Neg P\}) \in N \Longrightarrow simplify\ N\ (N - \{A + \{\#Pos P\} + \{\#Neg P\}\})$

condensation:

$(A + \{\#L\} + \{\#L\}) \in N \Longrightarrow simplify\ N\ (N - \{A + \{\#L\} + \{\#L\}\} \cup \{A + \{\#L\}\})$

subsumption:

$A \in N \Longrightarrow A \subset \# B \Longrightarrow B \in N \Longrightarrow simplify\ N\ (N - \{B\})$

```

lemma simplify-preserves-un-sat':
  fixes  $N N' :: 'v \text{ clauses}$ 
  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-clss-singleton true-clss-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-clss-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  $I \models_s N - \{B\} \Longrightarrow I \models A$  using  $\langle A \in N \rangle$  by (simp add: true-clss-def)
  moreover have  $I \models A \Longrightarrow I \models B$  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  $N N' :: 'v \text{ clauses}$ 
  assumes simplify  $N N'$ 
  and total-over-m  $I N$ 
  shows  $I \models_s N \longrightarrow I \models_s N'$ 
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce

```

```

lemma simplify-preserves-un-sat'':
  fixes  $N N' :: 'v \text{ clauses}$ 
  assumes simplify  $N N'$ 
  and total-over-m  $I N'$ 
  shows  $I \models_s N \longrightarrow I \models_s N'$ 
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce

```

```

lemma simplify-preserves-un-sat-eq:
  fixes  $N N' :: 'v \text{ clauses}$ 
  assumes simplify  $N N'$ 
  and total-over-m  $I N$ 
  shows  $I \models_s N \longleftrightarrow I \models_s N'$ 
  using simplify-preserves-un-sat simplify-preserves-un-sat' assms by blast

```

```

lemma simplify-preserves-finite:
  assumes simplify  $\psi \psi'$ 
  shows finite  $\psi \longleftrightarrow \text{finite } \psi'$ 
  using assms by (induct rule: simplify.induct, auto simp add: remove-def)

```

```

lemma rtranclp-simplify-preserves-finite:
  assumes rtranclp simplify  $\psi \psi'$ 

```

shows $\text{finite } \psi \longleftrightarrow \text{finite } \psi'$
using *assms* **by** (*induct rule: rtranclp-induct*) (*auto simp add: simplify-preserves-finite*)

lemma *simplify-atms-of-ms*:

assumes *simplify* $\psi \psi'$
shows $\text{atms-of-ms } \psi' \subseteq \text{atms-of-ms } \psi$
using *assms* **unfolding** *atms-of-ms-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 *rtranclp-simplify-atms-of-ms*:

assumes *rtranclp simplify* $\psi \psi'$
shows $\text{atms-of-ms } \psi' \subseteq \text{atms-of-ms } \psi$
using *assms* **apply** (*induct rule: rtranclp-induct*)
apply (*fastforce intro: simplify-atms-of-ms*)
using *simplify-atms-of-ms* **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 $'v \text{ uncon-state} = 'v \text{ clauses}$

inductive *uncon-res* :: $'v \text{ uncon-state} \Rightarrow 'v \text{ uncon-state} \Rightarrow \text{bool}$ **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 *rtranclp-uncon-inference-increasing*:

assumes *rtranclp uncon-res* $S S'$ **and** $\psi \in S$

shows $\psi \in S'$

using *assms* **by** (*induct rule: rtranclp-induct*) (*auto simp add: uncon-res-increasing*)

12.2.1 Subsumption

definition *subsumes* :: 'a literal multiset \Rightarrow 'a literal multiset \Rightarrow bool **where**

subsumes χ χ' \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 χ χ
unfolding *subsumes-def* **by** *auto*

lemma *subsumes-subsumption*:

assumes *subsumes* D χ
and $C \subset\# D$ **and** $\neg \text{tautology } \chi$
shows *subsumes* C χ **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* $(C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \})$ χ
shows *tautology* χ
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
 $\Longrightarrow \text{inference-clause } (N, \text{already-used}) (C + D, \text{already-used} \cup \{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\}) \mid$
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow \text{inference-clause } (N, \text{already-used}) (C + \{\#L\#\}, \text{already-used})$

inductive *inference* :: 'v state \Rightarrow 'v state \Rightarrow bool **where**

inference-step: *inference-clause* S (*clause*, *already-used*)
 $\Longrightarrow \text{inference } S$ (*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

$(\forall (A, B) \in \text{snd state}. \exists p. \text{Pos } p \in\# A \wedge \text{Neg } p \in\# B \wedge$
 $((\exists \chi \in \text{fst state}. \text{subsumes } \chi ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$
 $\vee \text{tautology } ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

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 rtranclp-inference-preserves-already-used-inv:
  assumes rtranclp inference  $S S'$ 
  and already-used-inv  $S$ 
  shows already-used-inv  $S'$ 
  using assms apply (induct rule: rtranclp-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  $a \in A$  and  $a0 \subset\# a$ 
    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  $A = \text{this}(1)$  and  $AB = \text{this}(2)$  and  $B = \text{this}(3)$  and  $\text{inv} = \text{this}(4)$ 
show ?case
  proof (standard, standard)
    fix  $x a b$ 
    assume  $x: x \in \text{snd } (N - \{B\}, \text{already-used})$  and [simp]:  $x = (a, b)$ 
    obtain  $p$  where  $p: \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$  and
       $q: (\exists \chi \in N. \text{subsumes } \chi (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \})))$ 
       $\vee \text{tautology } (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \}))$ 
    using inv  $x$  by fastforce
    consider (taut)  $\text{tautology } (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \})) \mid$ 
       $(\chi) \chi$  where  $\chi \in N$  subsumes  $\chi (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \}))$ 
       $\neg \text{tautology } (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \}))$ 
    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  $H = \text{this}$ 
    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  $C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\} \in N$ 
  assume already-used-inv (N, already-used)
  and (a, b)  $\in \text{snd } (N - \{C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\}, \text{already-used})$ 
  then obtain p where
     $\text{Pos } p \in \# a \wedge \text{Neg } p \in \# b \wedge$ 
     $((\exists \chi \in \text{fst } (N \cup \{C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\}, \text{already-used}).$ 
     $\text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
     $\vee \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
  by fastforce
  moreover have tautology (C + {#Pos P} + {#Neg P}) by auto
  ultimately show
     $\exists p. \text{Pos } p \in \# a \wedge \text{Neg } p \in \# b$ 
     $\wedge ((\exists \chi \in \text{fst } (N - \{C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\}, \text{already-used}).$ 
     $\text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
     $\vee \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
    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 $I \implies I \models \{\# \text{Pos } p\} + C \implies I \models \{\# \text{Neg } p\} + D \implies I \models C + D$ **and**

factoring-same-vars: $\text{atms-of } (\{\#L\} + \{\#L\} + C) = \text{atms-of } (\{\#L\} + C)$

unfolding true-cls-def consistent-interp-def **by** (fastforce split: split-if-asm)+

lemma inference-increasing:

assumes inference S S' **and** $\psi \in \text{fst } S$

shows $\psi \in \text{fst } S'$

using assms **by** (induct rule: inference.induct, auto)

lemma rtranclp-inference-increasing:

assumes rtranclp inference S S' **and** $\psi \in \text{fst } S$

shows $\psi \in \text{fst } S'$

using assms **by** (induct rule: rtranclp-induct, auto simp add: inference-increasing)

lemma *inference-clause-already-used-increasing*:
assumes *inference-clause* $S S'$
shows $\text{snd } S \subseteq \text{snd } S'$
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-ms*:
assumes *inference-clause* $S S'$
shows $\text{atms-of-ms } (\text{fst } (\text{fst } S \cup \{\text{fst } S'\}, \text{snd } S')) \subseteq \text{atms-of-ms } (\text{fst } S)$
using *assms* **apply** (*induct rule: inference-clause.induct*)
apply *auto*
apply (*metis Set.set-insert UnCI atms-of-ms-insert atms-of-plus*)
apply (*metis Set.set-insert UnCI atms-of-ms-insert atms-of-plus*)
apply (*simp add: in-m-in-literals union-assoc*)
unfolding *atms-of-ms-def* **using** *assms* **by** *fastforce*

lemma *inference-preserves-atms-of-ms*:
fixes $N N' :: 'v \text{ clauses}$
assumes *inference* $T T'$
shows $\text{atms-of-ms } (\text{fst } T') \subseteq \text{atms-of-ms } (\text{fst } T)$
using *assms* **apply** (*induct rule: inference.induct*)
using *inference-clause-preserves-atms-of-ms* **by** *fastforce*

lemma *inference-preserves-total*:
fixes $N N' :: 'v \text{ clauses}$
assumes *inference* $(N, \text{already-used}) (N', \text{already-used}')$
shows $\text{total-over-m } I N \implies \text{total-over-m } I N'$
using *assms* *inference-preserves-atms-of-ms* **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference T T'*
shows *total-over-m I (fst T) \implies total-over-m I (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 fst N \longleftrightarrow I \models_s 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 ψ ψ' and finite (fst ψ)*
shows *finite (fst ψ')*
using *assms* **by** (*induct rule: inference.induct, auto simp add: simplify-preserves-finite*)

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause ψ ψ' and finite (snd ψ)*
shows *finite (snd ψ')*
using *assms* **by** (*induct rule: inference-clause.induct, auto*)

lemma *inference-preserves-finite-snd*:
assumes *inference ψ ψ' and finite (snd ψ)*
shows *finite (snd ψ')*
using *assms inference-clause-preserves-finite-snd* **by** (*induct rule: inference.induct, fastforce*)

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference ψ ψ' and finite (fst ψ)*
shows *finite (fst ψ')*
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 ψ ψ'*
and *satisfiable ψ'*
shows *satisfiable ψ*
using *assms*

proof *induction*

case (*tautology-deletion* A P) **note** $AP = \text{this}(1)$ **and** $\text{sat} = \text{this}(2)$
let $?A' = A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}$
let $? \psi' = \psi - \{?A'\}$
obtain I **where**
 $I: I \models_s ? \psi'$ **and**
 $\text{cons: consistent-interp } I$ **and**
 $\text{tot: total-over-m } I ? \psi'$
using sat **unfolding** *satisfiable-def* **by** *auto*
{ assume $Pos\ P \in I \vee Neg\ P \in I$
then have $I \models ?A'$ **by** *auto*
then have $I \models_s \psi$ **using** I **by** (*metis insert-Diff tautology-deletion.hyps true-clss-insert*)
then have $?case$ **using** cons tot **by** *auto*
}
moreover {
assume $Pos: Pos\ P \notin I$ **and** $Neg: Neg\ P \notin I$
then have *consistent-interp* $(I \cup \{Pos\ P\})$ **using** cons **by** *simp*
moreover have $I'A: I \cup \{Pos\ P\} \models ?A'$ **by** *auto*
have $\{Pos\ P\} \cup I \models_s \psi - \{A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\}$
using $\langle I \models_s \psi - \{A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\rangle$ *true-clss-union-increase'* **by** *blast*
then have $I \cup \{Pos\ P\} \models_s \psi$
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** $AL = \text{this}(1)$ **and** $\text{sat} = \text{this}(2)$
have $f3: \text{simplify } \psi (\psi - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\})$
using AL *simplify.condensation* **by** *blast*
obtain $LL :: 'a \text{ literal multiset set} \Rightarrow 'a \text{ literal set}$ **where**
 $f4: LL (\psi - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\}) \models_s \psi - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\}$
 $\wedge \text{consistent-interp } (LL (\psi - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\}))$
 $\wedge \text{total-over-m } (LL (\psi - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\})) (\psi - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\})$
using sat **by** (*meson satisfiable-def*)
have $f5: \text{insert } (A + \{\#L\# \} + \{\#L\# \}) (\psi - \{A + \{\#L\# \} + \{\#L\# \}\}) = \psi$
using AL **by** *fastforce*
have $\text{atms-of } (A + \{\#L\# \} + \{\#L\# \}) = \text{atms-of } (\{\#L\# \} + A)$
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** $A = \text{this}(1)$ **and** $AB = \text{this}(2)$ **and** $B = \text{this}(3)$ **and** $\text{sat} = \text{this}(4)$
let $? \psi' = \psi - \{B\}$
obtain I **where** $I: I \models_s ? \psi'$ **and** $\text{cons: consistent-interp } I$ **and** $\text{tot: total-over-m } I ? \psi'$
using sat **unfolding** *satisfiable-def* **by** *auto*
have $I \models A$ **using** $A\ I$ **by** (*metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def*)
then have $I \models B$ **using** AB *subset-mset.less-imp-le true-clss-mono-leD* **by** *blast*
then have $I \models_s \psi$ **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  $\psi$   $\psi'$ 
  shows satisfiable (fst  $\psi'$ )  $\longrightarrow$  satisfiable (fst  $\psi$ )
  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: rtranclp-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]:
  ( $\bigwedge xs:: 'v$  sem-tree. ( $\bigwedge ys:: 'v$  sem-tree. sem-tree-size ys < sem-tree-size xs  $\Longrightarrow$  P ys)  $\Longrightarrow$  P xs)
   $\Longrightarrow$  P 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  $\psi \longleftrightarrow$ 
    (partial-interps ag (I  $\cup$  {Pos v})  $\psi \wedge$  partial-interps ad (I  $\cup$  {Neg v})  $\psi$ )

lemma simplify-preserve-partial-leaf:
  simplify  $N$   $N' \Longrightarrow$  partial-interps Leaf I  $N \Longrightarrow$  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-ms-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 rtranclp-inference-preserve-partial-tree:
  assumes rtranclp inference N N'
  and partial-interps t I (fst N)
  shows partial-interps t I (fst N')
  using assms apply (induct rule: rtranclp-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 ( $\lambda(A, -). \text{card } A$ ), 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-ms-def by blast

lemma partial-interps-build-sem-tree-atms-general:
  fixes  $\psi :: 'v :: \text{linorder clauses}$  and  $p :: 'v \text{ literal list}$ 
  assumes unsat: unsatisfiable  $\psi$  and finite  $\psi$  and consistent-interp I
  and finite atms
  and atms-of-ms  $\psi$  = 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-ms  $\psi$  = 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:  $\bigwedge \chi. \chi \in \psi \implies \text{total-over-m Ia } \{\chi\}$ 
        unfolding total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def
        using atmsIa atms-of-ms-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-ms-def)

    ultimately have ?case by metis
  }
moreover {

```

```

assume atms: atms ≠ {}
have build-sem-tree atms  $\psi = \text{Node } (\text{Min } \textit{atms}) (\text{build-sem-tree } (\text{Set.remove } (\text{Min } \textit{atms}) \textit{atms}) \psi)$ 
  (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  $\cup \{\text{Pos } (\text{Min } \textit{atms})\}$ ) 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-ms  $\psi = \text{Set.remove } (\text{Min } \textit{atms}) \textit{atms} \cup \textit{atms-of-s } (\text{Ia} \cup \{\text{Pos } (\text{Min } \textit{atms})\})$ 
  using Min-in atms f un by fastforce
moreover have disj': Set.remove (Min atms) atms  $\cap \textit{atms-of-s } (\text{Ia} \cup \{\text{Pos } (\text{Min } \textit{atms})\}) = \{\}$ 
  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  $\cup \{\text{Pos } (\text{Min } \textit{atms})\}$ )  $\psi$ 
using IH1[of Ia  $\cup \{\text{Pos } (\text{Min } (\textit{atms}))\}$ ] atms f unsat finite by metis

have consistent-interp (Ia  $\cup \{\text{Neg } (\text{Min } \textit{atms})\}$ ) 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-ms  $\psi = \text{Set.remove } (\text{Min } \textit{atms}) \textit{atms} \cup \textit{atms-of-s } (\text{Ia} \cup \{\text{Neg } (\text{Min } \textit{atms})\})$ 
  using  $\langle \textit{atms-of-ms } \psi = \text{Set.remove } (\text{Min } \textit{atms}) \textit{atms} \cup \textit{atms-of-s } (\text{Ia} \cup \{\text{Pos } (\text{Min } \textit{atms})\}) \rangle$  by
blast

moreover have disj': Set.remove (Min atms) atms  $\cap \textit{atms-of-s } (\text{Ia} \cup \{\text{Neg } (\text{Min } \textit{atms})\}) = \{\}$ 
  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  $\cup \{\text{Neg } (\text{Min } \textit{atms})\}$ )  $\psi$ 
using IH2[of Ia  $\cup \{\text{Neg } (\text{Min } (\textit{atms}))\}$ ] 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-ms  $\psi$ )  $\psi$ )  $\{\}$   $\psi$ 
proof –
  have consistent-interp  $\{\}$  unfolding consistent-interp-def by auto
  moreover have atms-of-ms  $\psi = \textit{atms-of-ms } \psi \cup \textit{atms-of-s } \{\}$  unfolding atms-of-s-def by auto
  moreover have atms-of-ms  $\psi \cap \textit{atms-of-s } \{\} = \{\}$  unfolding atms-of-s-def by auto
  moreover have finite (atms-of-ms  $\psi$ ) unfolding atms-of-ms-def using finite by simp
  ultimately show partial-interps (build-sem-tree (atms-of-ms  $\psi$ )  $\psi$ )  $\{\}$   $\psi$ 
    using partial-interps-build-sem-tree-atms-general[of  $\psi \{\}$  atms-of-ms  $\psi$ ] 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  $n \chi$ )
  note  $IH = \text{this}(1)$  and  $\text{count} = \text{this}(2)$  and  $L = \text{this}(3)$  and  $\chi = \text{this}(4)$ 
  {
    assume  $n = 0$ 
    then have inference**  $\psi \psi$ 
    and  $\chi \in \text{fst } \psi$ 
    and  $\forall L. (L \in \# \chi) \longleftrightarrow (L \in \# \chi)$ 
    and  $\text{count } \chi L = (1::\text{nat})$ 
    and  $\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi$ 
    by (auto simp add: count L  $\chi$ )
    then have ?case by metis
  }
  moreover {
    assume  $n > 0$ 
    then have  $\exists C. \chi = C + \{\#L, L\# \}$ 
    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  $C: \chi = C + \{\#L, L\# \}$  by metis
    let  $? \chi' = C + \{\#L\# \}$ 
    let  $? \psi' = (\text{fst } \psi \cup \{? \chi'\}, \text{snd } \psi)$ 
    have  $\varphi: \forall \varphi \in \text{fst } \psi. (\varphi \in \text{fst } \psi \vee \varphi \neq ? \chi') \longleftrightarrow \varphi \in \text{fst } ? \psi'$  unfolding  $C$  by auto
    have inf: inference  $\psi ? \psi'$ 
    using  $C$  factoring  $\chi$  prod.collapse union-commute inference-step by metis
    moreover have  $\text{count}' : \text{count } ? \chi' L = n$  using  $C$  count by auto
    moreover have  $L \chi' : L : \# ? \chi'$  by auto
    moreover have  $\chi' \psi' : ? \chi' \in \text{fst } ? \psi'$  by auto
    ultimately obtain  $\psi''$  and  $\chi''$ 
    where
    inference**  $? \psi' \psi''$  and
     $\alpha: \chi'' \in \text{fst } \psi''$  and
     $\forall La. (La \in \# ? \chi') \longleftrightarrow (La \in \# \chi'')$  and
     $\beta: \text{count } \chi'' L = (1::\text{nat})$  and
     $\varphi': \forall \varphi. \varphi \in \text{fst } ? \psi' \longrightarrow \varphi \in \text{fst } \psi''$  and
     $I \chi: I \models ? \chi' \longleftrightarrow I \models \chi''$  and
     $\text{tot}: \forall I'. \text{total-over-m } I' \{? \chi'\} \longrightarrow \text{total-over-m } I' \{\chi''\}$ 
    using  $IH[\text{of } ? \chi' ? \psi']$  count'  $L \chi' \chi' \psi'$  by blast

    then have inference**  $\psi \psi''$ 
    and  $\forall La. (La \in \# \chi) \longleftrightarrow (La \in \# \chi'')$ 
    using inf unfolding C by auto
    moreover have  $\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi''$  using  $\varphi \varphi'$  by metis
    moreover have  $I \models \chi \longleftrightarrow I \models \chi''$  using  $I \chi$  unfolding true-cls-def C by auto
    moreover have  $\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi''\}$ 
  }

```



```

    using tot unfolding C total-over-m-def by auto
    ultimately have ?case using  $\varphi \varphi' \alpha \beta$  by metis
  }
  ultimately show ?case by auto
qed

lemma can-decrease-tree-size:
  fixes  $\psi :: 'v$  state and tree ::  $'v$  sem-tree
  assumes finite (fst  $\psi$ ) and already-used-inv  $\psi$ 
  and partial-interps tree I (fst  $\psi$ )
  shows  $\exists (tree' :: 'v$  sem-tree)  $\psi'. inference^{**} \psi \psi' \wedge partial-interps tree' I (fst \psi')$ 
     $\wedge (sem-tree-size tree' < sem-tree-size tree \vee sem-tree-size tree = 0)$ 
  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
        tot $\chi$ : total-over-m (I  $\cup \{Pos\ v\}$ )  $\{\chi\}$  and
         $\chi\psi$ :  $\chi \in fst\ \psi$  and
         $\chi': \neg I \cup \{Neg\ v\} \models \chi'$  and
        tot $\chi'$ : total-over-m (I  $\cup \{Neg\ v\}$ )  $\{\chi'\}$  and
         $\chi'\psi$ :  $\chi' \in fst\ \psi$ 
        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 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 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: rtrancplp 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 $\chi$ :  $\forall I'. \text{total-over-m } I' \{ \chi \} \longrightarrow \text{total-over-m } I' \{ \chi^2 \}$ 
    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  $\chi^{2'} \text{-incl}$ :  $\chi^{2'} \in \text{fst } \psi''$ 
    and  $\chi' \chi^{2'} \text{-incl}$ :  $\forall L :: 'v$  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 $\chi'$ :  $\forall I'. \text{total-over-m } I' \{ \chi' \} \longrightarrow \text{total-over-m } I' \{ \chi^{2'} \}$ 
    using can-decrease-count[of  $\chi' \text{ Pos v count } \chi' \text{ (Pos v) } \psi' \text{ I}$ ] by auto

  obtain C where  $\chi^2$ :  $\chi^2 = C + \{ \# \text{Neg v} \# \}$  and negC: Neg v  $\notin \# C$  and posC: Pos v  $\notin \# C$ 
    by (metis (no-types, lifting) One-nat-def Posv Suc-inject Suc-pred  $\chi \chi^2 \text{-incl}$  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
    posC': Pos v  $\notin \# C'$  and
    negC': Neg v  $\notin \# C'$ 
  proof -
    assume a1:  $\bigwedge C'. \llbracket \chi^{2'} = C' + \{ \# \text{Pos v} \# \}; \text{Pos v} \notin \# C'; \text{Neg v} \notin \# C' \rrbracket \implies \text{thesis}$ 
    have f2:  $\bigwedge n. (n :: \text{nat}) - n = 0$ 
      by simp
    have Neg v  $\notin \# \chi^{2'} - \{ \# \text{Pos v} \# \}$ 
      using Negv  $\chi' \chi^{2'} \text{-incl}$  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 totC: total-over-m I {C}
    using tot-imp $\chi$  tot $\chi$  tot-over-m-remove[of I Pos v C] negC posC unfolding  $\chi^2$ 

```

```

    by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
  have totC': total-over-m I {C'}
    using tot-impχ' totχ' total-over-m-sum tot-over-m-remove[of I Neg v C'] negC' posC'
    unfolding χ2' by (metis total-over-m-sum uminus-Neg)
  have ¬ I ⊨ C + C'
    using χ Iχ χ' Iχ' unfolding χ2 χ2' true-cls-def Bex-mset-def
    by (metis add-gr-0 count-union true-cls-singleton true-cls-union-increase)
  then have part-I-ψ''': partial-interps Leaf I (fst ψ'' ∪ {C + C'})
    using totC totC' by simp
    (metis ¬ I ⊨ C + C' atms-of-ms-singleton total-over-m-def total-over-m-sum)
  {
    assume ({#Pos v#} + C', {#Neg v#} + C) ∉ snd ψ''
    then have inf'': inference ψ'' (fst ψ'' ∪ {C + C'}, snd ψ'' ∪ {(χ2', χ2)})
      using add commute φ' χ2incl χ2' ∈ fst ψ'' unfolding χ2 χ2'
      by (metis prod.collapse inference-step resolution)
    have inference** ψ (fst ψ'' ∪ {C + C'}, snd ψ'' ∪ {(χ2', χ2)})
      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-ψ''' by (metis fst-conv)
  }
  moreover {
    assume a: ({#Pos v#} + C', {#Neg v#} + C) ∈ snd ψ''
    then have (∃χ ∈ fst ψ''. (∀I. total-over-m I {C+C'} → total-over-m I {χ})
      ∧ (∀I. total-over-m I {χ} → I ⊨ χ → I ⊨ C' + C))
      ∨ tautology (C' + C)
    proof -
      obtain p where p: Pos p ∈# ({#Pos v#} + C') and
        n: Neg p ∈# ({#Neg v#} + C) and
        decomp: ((∃χ ∈ fst ψ''.
          (∀I. total-over-m I ({#Pos v#} + C') - {#Pos p#}
            + (({#Neg v#} + C) - {#Neg p#}))
            → total-over-m I {χ})
          ∧ (∀I. total-over-m I {χ} → I ⊨ χ
            → I ⊨ ({#Pos v#} + C') - {#Pos p#} + (({#Neg v#} + C) - {#Neg p#})))
          ∨ tautology (({#Pos v#} + C') - {#Pos p#} + (({#Neg v#} + C) - {#Neg p#})))
      using a by (blast intro: allE[OF a-u-i-ψ''[unfolded subsumes-def Ball-def],
        of ({#Pos v#} + C', {#Neg v#} + C)])
    { assume p ≠ v
      then have Pos p ∈# C' ∧ Neg p ∈# 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 ∃χ ∈ fst ψ''. (∀I. total-over-m I {C+C'} → total-over-m I {χ})
      ∧ (∀I. total-over-m I {χ} → I ⊨ χ → I ⊨ C' + C)
    then obtain ∅ where ∅: ∅ ∈ fst ψ'' and
      tot-∅-CC': ∀I. total-over-m I {C+C'} → total-over-m I {∅} and
      ∅-inv: ∀I. total-over-m I {∅} → I ⊨ ∅ → I ⊨ C' + C by blast
    have partial-interps Leaf I (fst ψ'')

```

```

    using tot- $\vartheta$ -CC'  $\vartheta$   $\vartheta$ -inv totC totC'  $\langle \neg I \models C + C' \rangle$  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  $\neg$ tautology (C' + C)
      using  $\langle \neg I \models C + C' \rangle$  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  $\cup$  {Pos v}) (fst  $\psi$ )
    and partad: partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ag < sem-tree-size xs  $\longrightarrow$  finite (fst  $\psi$ )  $\longrightarrow$  already-used-inv  $\psi$ 
     $\longrightarrow$  ( partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi$ )  $\longrightarrow$ 
      ( $\exists$  tree'  $\psi'$ . inference**  $\psi$   $\psi'$   $\wedge$  partial-interps tree' (I  $\cup$  {Pos v}) (fst  $\psi'$ )
         $\wedge$  (sem-tree-size tree' < sem-tree-size ag  $\vee$  sem-tree-size ag = 0)))
    using IH by auto
  ultimately obtain  $\psi' :: 'v$  state and tree' :: 'v sem-tree where
    inf: inference**  $\psi$   $\psi'$ 
    and part: partial-interps tree' (I  $\cup$  {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 by blast

  have partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi'$ )
    using rtranclp-inference-preserve-partial-tree inf partad by metis
  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  $\cup$  {Pos v}) (fst  $\psi$ ) and
    partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ad < sem-tree-size xs  $\longrightarrow$  finite (fst  $\psi$ )  $\longrightarrow$  already-used-inv  $\psi$ 
     $\longrightarrow$  ( partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
       $\longrightarrow$  ( $\exists$  tree'  $\psi'$ . inference**  $\psi$   $\psi'$   $\wedge$  partial-interps tree' (I  $\cup$  {Neg v}) (fst  $\psi'$ )
         $\wedge$  (sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0)))
    using IH by auto
  ultimately obtain  $\psi' :: 'v$  state and tree' :: 'v sem-tree where
    inf: inference**  $\psi$   $\psi'$ 
    and part: partial-interps tree' (I  $\cup$  {Neg v}) (fst  $\psi'$ )
    and size: sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0

```

```

    using finite part rtrancp.rtranc1-refl a-u-i by blast

  have partial-interps ag ( $I \cup \{Pos\ v\}$ ) (fst  $\psi'$ )
    using rtrancp-inference-preserve-partial-tree inf partag by metis
  then have partial-interps (Node v ag tree') I (fst  $\psi'$ ) 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  $\psi :: 'v :: linorder\ state$ 
  assumes
    unsat:  $\neg$ satisfiable (fst  $\psi$ ) and
    finite: finite (fst  $\psi$ ) and
    a-u-v: already-used-inv  $\psi$ 
  shows  $\exists \psi'. (inference^{**} \psi \psi' \wedge \{\#\} \in fst \psi')$ 
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  $H = this$ 
    {
      fix  $\chi$ 
      assume tree: tree = Leaf
      obtain  $\chi$  where  $\chi: \neg \{\} \models \chi$  and tot $\chi$ : total-over-m {} { $\chi$ } and  $\chi\psi: \chi \in fst \psi$ 
        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 $^{**} \psi \psi$  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 $^{**} \psi \psi'$  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 rtrancp-inference-preserves-finite inf H(4) by metis
    moreover have unsatisfiable (fst  $\psi'$ )
      using inference-preserves-unsat inf bigger.prem(2) by blast
    moreover have already-used-inv  $\psi'$ 
      using H(5) inf rtrancp-inference-preserves-already-used-inv[of  $\psi \psi'$ ] by auto
    ultimately have ?case using inf rtrancp-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* ψ **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-clss-def*)

lemma *inference-soundness-and-completeness*:
fixes $\psi :: 'v :: \text{linorder state}$
assumes *finite*: *finite (fst } \psi)*
and *snd* $\psi = \{\}$
shows $(\exists \psi'. (\text{inference}^{**} \ \psi \ \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable (fst } \psi)$
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* ψ **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 χ' : $? \chi' + \{\#L\# \} + \{\#L\# \} = \chi$
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* ψ **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  $L \in \# \chi \wedge - L \in \# \chi$  by metis
then obtain  $\chi'$  where  $\chi = \chi' + \{\#Pos\ (atm-of\ L)\# \} + \{\#Neg\ (atm-of\ L)\# \}$ 
  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  $\sim$  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  $\exists l'. \textit{simplify}\ \{\psi\}\ l'$ 
    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-cls-def apply auto
  using assms true-cls-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* *trancplD*)

12.5 Resolution and Invariants

inductive *resolution* :: 'v state \Rightarrow 'v state \Rightarrow bool **where**

full1-simp: *full1 simplify* $N\ N' \Longrightarrow \text{resolution } (N, \text{already-used})\ (N', \text{already-used})$ |

inferring: *inference* $(N, \text{already-used})\ (N', \text{already-used}') \Longrightarrow \text{simplified } N$

$\Longrightarrow \text{full simplify } N'\ N'' \Longrightarrow \text{resolution } (N, \text{already-used})\ (N'', \text{already-used}')$

12.5.1 Invariants

lemma *resolution-finite*:

assumes *resolution* $\psi\ \psi'$ **and** *finite* (*fst* ψ)

shows *finite* (*fst* ψ')

using *assms* **by** (*induct* rule: *resolution.induct*)

(*auto simp add*: *full1-def full-def rtrancpl-simplify-preserves-finite*

dest: *trancpl-into-rtrancpl inference-preserves-finite*)

lemma *rtrancpl-resolution-finite*:

assumes *resolution*** $\psi\ \psi'$ **and** *finite* (*fst* ψ)

shows *finite* (*fst* ψ')

using *assms* **by** (*induct* rule: *rtrancpl-induct*, *auto simp add*: *resolution-finite*)

lemma *resolution-finite-snd*:

assumes *resolution* $\psi\ \psi'$ **and** *finite* (*snd* ψ)

shows *finite* (*snd* ψ')

using *assms* **apply** (*induct* rule: *resolution.induct*, *auto simp add*: *inference-preserves-finite-snd*)

using *inference-preserves-finite-snd snd-conv* **by** *metis*

lemma *rtrancpl-resolution-finite-snd*:

assumes *resolution*** $\psi\ \psi'$ **and** *finite* (*snd* ψ)

shows *finite* (*snd* ψ')

using *assms* **by** (*induct* rule: *rtrancpl-induct*, *auto simp add*: *resolution-finite-snd*)

lemma *resolution-always-simplified*:

assumes *resolution* $\psi\ \psi'$

shows *simplified* (*fst* ψ')

using *assms* **by** (*induct* rule: *resolution.induct*)

(*auto simp add*: *full1-def full-def*)

lemma *trancpl-resolution-always-simplified*:

assumes *trancpl resolution* $\psi\ \psi'$

shows *simplified* (*fst* ψ')

using *assms* **by** (*induct* rule: *trancpl.induct*, *auto simp add*: *resolution-always-simplified*)

lemma *resolution-atms-of*:

assumes *resolution* $\psi\ \psi'$ **and** *finite* (*fst* ψ)

shows *atms-of-ms* (*fst* ψ') \subseteq *atms-of-ms* (*fst* ψ)

using *assms* **apply** (*induct* rule: *resolution.induct*)

apply(*simp add*: *rtrancpl-simplify-atms-of-ms trancpl-into-rtrancpl full1-def*)

by (*metis* (*no-types*, *lifting*) *contra-subsetD fst-conv full-def*

inference-preserves-atms-of-ms rtrancpl-simplify-atms-of-ms subsetI)

lemma *rtrancpl-resolution-atms-of*:

assumes *resolution*** $\psi\ \psi'$ **and** *finite* (*fst* ψ)

shows *atms-of-ms* (*fst* ψ') \subseteq *atms-of-ms* (*fst* ψ)

using *assms* **apply** (*induct rule: rtrancpl-induct*)
using *resolution-atms-of rtrancpl-resolution-finite* **by** *blast+*

lemma *resolution-include:*

assumes *res: resolution $\psi \psi'$ and finite: finite (fst ψ)*
shows *fst $\psi' \subseteq \text{build-all-simple-clss (atms-of-ms (fst } \psi))$*

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 *fst $\psi' \subseteq \text{build-all-simple-clss (atms-of-ms (fst } \psi'))$*
using *simplified-in-build-all finite' simplified-imp-distinct-mset-tauto[of fst ψ']* **by** *auto*
moreover have *atms-of-ms (fst $\psi') \subseteq \text{atms-of-ms (fst } \psi)$*
using *res finite resolution-atms-of[of $\psi \psi'$]* **by** *auto*
ultimately show *?thesis* **by** (*meson atms-of-ms-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 ψ)*
shows *fst $\psi' \subseteq \text{build-all-simple-clss (atms-of-ms (fst } \psi))$*
using *assms* **apply** (*induct rule: trancpl.induct*)
apply (*simp add: resolution-include*)
by (*meson atms-of-ms-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 literal multiset \times 'a literal multiset) set \Rightarrow 'a set \Rightarrow bool **where**

already-used-all-simple already-used vars \equiv

($\forall (A, B) \in \text{already-used. simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars}$)

lemma *already-used-all-simple-vars-incl:*

assumes *vars \subseteq vars'*
shows *already-used-all-simple a vars \implies already-used-all-simple a vars'*
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 *atms-of-ms (fst S) \subseteq vars*
shows *already-used-all-simple (snd (fst $S \cup \{\text{fst } S'\}, \text{snd } S'))$ 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 = this*

show *?case* **apply** *clarify*

proof –

fix *A B v*

assume *(A, B) \in snd (fst (N , 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 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 eq:  $(A, B) = (\{\#Pos\ P\# \} + C, \{\#Neg\ P\# \} + D)$ 
    then have  $\text{simplified } \{A\}$  using  $\text{simplified-in } H(1,5)$  by auto
    moreover have  $\text{simplified } \{B\}$  using eq  $\text{simplified-in } H(2,5)$  by auto
    moreover have  $\text{atms-of } A \subseteq \text{atms-of-ms } N$ 
      using eq  $H(1)$   $\text{atms-of-atms-of-ms-mono}[of\ A\ N]$  by auto
    moreover have  $\text{atms-of } B \subseteq \text{atms-of-ms } N$ 
      using eq  $H(2)$   $\text{atms-of-atms-of-ms-mono}[of\ B\ N]$  by auto
    ultimately 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  $H(6)$  by auto
  }
  ultimately show  $\text{simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars}$ 
    by fast
qed

```

lemma *inference-preserves-already-used-all-simple:*
 assumes *inference* $S\ S'$
 and *already-used-all-simple* $(\text{snd } S)\ \text{vars}$
 and *simplified* $(\text{fst } S)$
 and $\text{atms-of-ms } (\text{fst } S) \subseteq \text{vars}$
 shows *already-used-all-simple* $(\text{snd } S')\ \text{vars}$
 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* $[of\ S\ (\text{clause}, \text{already-used})\ \text{vars}]$
 by auto
qed

lemma *already-used-all-simple-inv:*
 assumes *resolution* $S\ S'$
 and *already-used-all-simple* $(\text{snd } S)\ \text{vars}$
 and $\text{atms-of-ms } (\text{fst } S) \subseteq \text{vars}$
 shows *already-used-all-simple* $(\text{snd } S')\ \text{vars}$
 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* $(\text{snd } (N'', \text{already-used'}))\ \text{vars}$
 using *inference-preserves-already-used-all-simple* $[of\ (N, \text{already-used})]$ by *simp*
qed

lemma *rtrancplp-already-used-all-simple-inv:*
 assumes *resolution*** $S\ S'$
 and *already-used-all-simple* $(\text{snd } S)\ \text{vars}$
 and $\text{atms-of-ms } (\text{fst } S) \subseteq \text{vars}$
 and *finite* $(\text{fst } S)$
 shows *already-used-all-simple* $(\text{snd } S')\ \text{vars}$

```

using assms
proof (induct rule: rtranclp-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-ms (fst S')  $\subseteq$  atms-of-ms (fst S)
    by (simp add: infstar local.finite rtranclp-resolution-atms-of)
  then have atms-of-ms (fst S')  $\subseteq$  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 snd S  $\subset$  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 snd S  $\subset$  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 snd S  $\subset$  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 snd S  $\subset$  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 *build-all-simple-clss vars* \times *build-all-simple-clss vars*

```

lemma already-used-all-simple-in-already-used-top:
  assumes already-used-all-simple s vars and finite vars
  shows s  $\subseteq$  already-used-top vars
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** *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** *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
build-all-simple-clss-mono[of vars *atms-of* B] x *assms*(2) **by** *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 ψ) vars*
and *finite-v: finite vars*
and *finite-fst: finite (fst ψ)*
and *finite-snd: finite (snd ψ)*
and *simp: simplified (fst ψ)*
and *atms-of-ms (fst ψ) \subseteq vars*
shows *card-simple vars (snd ψ')* $<$ *card-simple vars (snd ψ)*

proof –

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 ψ)* $=$ *card ?top* $-$ *card (snd ψ)*
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 ψ') vars*
using *already-used-all-simple-inv res a-u-s assms*(7) **by** *blast*
have *f*: *finite (snd ψ')* **using** *already-used-all-simple-finite a-u-s' finite-v* **by** *auto*
have 2: *card-simple vars (snd ψ')* $=$ *card ?top* $-$ *card (snd ψ')*
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 ψ')*

```

    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 *tranclp-resolution-card-simple-decreasing:*

```

  assumes tranclp_resolution  $\psi \psi'$  and finite-fst: finite (fst  $\psi$ )
  and already-used-all-simple (snd  $\psi$ ) vars
  and atms-of-ms (fst  $\psi$ )  $\subseteq$  vars
  and finite-v: finite vars
  and finite-snd: finite (snd  $\psi$ )
  and simplified (fst  $\psi$ )
  shows card-simple vars (snd  $\psi'$ ) < card-simple vars (snd  $\psi$ )
  using assms
proof (induct rule: tranclp.induct)
  case (r-into-trancl  $\psi \psi'$ )
  then show ?case by (simp add: resolution-card-simple-decreasing)
next
  case (trancl-into-trancl  $\psi \psi' \psi''$ ) 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 rtranclp-already-used-all-simple-inv[OF tranclp-into-rtranclp[OF res] a-u-s atms f-fst] .
  have finite (fst  $\psi'$ )
    by (meson build-all-simple-clss-finite rev-finite-subset rtranclp-resolution-include
      trancl-into-trancl.hyps(1) trancl-into-trancl.prem(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 tranclp-resolution-always-simplified by blast
  moreover have atms-of-ms (fst  $\psi'$ )  $\subseteq$  vars
    by (meson atms f-fst order.trans res rtranclp-resolution-atms-of tranclp-into-rtranclp)
  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 *tranclp-resolution-card-simple-decreasing-2:*

```

  assumes tranclp_resolution  $\psi \psi'$ 
  and finite-fst: finite (fst  $\psi$ )
  and empty-snd: snd  $\psi$  = {}
  and simplified (fst  $\psi$ )
  shows card-simple (atms-of-ms (fst  $\psi$ )) (snd  $\psi'$ ) < card-simple (atms-of-ms (fst  $\psi$ )) (snd  $\psi$ )
proof -
  let ?vars = (atms-of-ms (fst  $\psi$ ))
  have already-used-all-simple (snd  $\psi$ ) ?vars unfolding empty-snd by auto
  moreover have atms-of-ms (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) tranclp-resolution-card-simple-decreasing[of  $\psi \psi'$ ] by presburger

```

qed

12.5.2 well-foundness if the relation

lemma *wf-simplified-resolution*:

assumes *f-vars*: *finite vars*

shows *wf* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (fst\ x) \subseteq \text{vars} \wedge \text{simplified } (fst\ x) \wedge \text{finite } (snd\ x) \wedge \text{finite } (fst\ x) \wedge \text{already-used-all-simple } (snd\ x)\ \text{vars}) \wedge \text{resolution } x\ y\}$

proof –

```
{
  fix a b :: 'v::linorder state
  assume (b, a) ∈ {(y, x). (atms-of-ms (fst x) ⊆ vars ∧ simplified (fst x) ∧ finite (snd x)
    ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
  then have
    atms-of-ms (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-ms (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-ms (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* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (fst\ x) \subseteq \text{vars} \wedge \neg \text{simplified } (fst\ x) \wedge \text{finite } (snd\ x) \wedge \text{finite } (fst\ x) \wedge \text{already-used-all-simple } (snd\ x)\ \text{vars}) \wedge \text{resolution } x\ y\}$

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* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (fst\ x) \subseteq \text{vars} \wedge \text{simplified } (fst\ x) \wedge \text{finite } (snd\ x) \wedge \text{finite } (fst\ x) \wedge \text{already-used-all-simple } (snd\ x)\ \text{vars}) \wedge \text{resolution } x\ y\} \cup \{(y, x). (\text{atms-of-ms } (fst\ x) \subseteq \text{vars} \wedge \neg \text{simplified } (fst\ x) \wedge \text{finite } (snd\ x) \wedge \text{finite } (fst\ x) \wedge \text{already-used-all-simple } (snd\ x)\ \text{vars}) \wedge \text{resolution } x\ y\}$ (**is** *wf* (*?R* \cup *?S*))

proof –

have *Domain ?R Int Range ?S* = {} **using** *resolution-always-simplified* **by** *auto blast*

then show *wf* (*?R* \cup *?S*)

using *wf-simplified-resolution*[*OF f-vars*] *wf-simplified-resolution'*[*OF f-vars*] *wf-Un*[*of ?R ?S*]

by *fast*

qed

```

lemma rtranclp-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 rtrancpl-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  $t$   $I$  (fst  $S$ )
  shows partial-interps  $t$   $I$  (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  $t$   $I$  (fst  $S$ )
  shows partial-interps  $t$   $I$  (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 |  $\varphi$  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 |  $\varphi$  L.  $(\varphi = x \vee \varphi \in N) \wedge L \in \# \varphi \wedge P \varphi L$ }  $\subseteq$  {f x L | L. L  $\in \# x \wedge P x L$ }
     $\cup$  {f  $\varphi$  L |  $\varphi$  L.  $\varphi \in N \wedge L \in \# \varphi \wedge P \varphi L$ } by auto
  moreover have finite {f x L | L. L  $\in \# x$ } by auto
  ultimately show ?case using IH finite-subset by fastforce
qed

value card
value filter-mset
value {#count  $\varphi$  L | 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 multiset  $\Rightarrow$  'a multiset
    (({#-/./ - : setof -#}))
translations
  {#e. x: setof M#} == CONST set-mset (CONST image-mset (%x. e) M)
value {# a. a : setof {#1,1,2::int#}#} = {1,2}

definition sum-count-ge-2 :: 'a multiset set  $\Rightarrow$  nat ( $\Xi$ ) where
  sum-count-ge-2  $\equiv$  folding.F ( $\lambda \varphi. \text{op} + (\text{msetsum } \{ \# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \# \})$ ) 0

interpretation sum-count-ge-2:
  folding ( $\lambda \varphi. \text{op} + (\text{msetsum } \{ \# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \# \})$ ) 0
rewrites
  folding.F ( $\lambda \varphi. \text{op} + (\text{msetsum } \{ \# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L \# \})$ ) 0 = 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
    = 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 $a \notin A$

then have $A \subseteq F$ using *AF* by *auto*

then show ?case using *IH[of A]* by (*simp add: aF local.finite*)

next

assume *aA: a ∈ A*

then have $A - \{a\} \subseteq F$ using *AF* by *auto*

then have $\Xi (A - \{a\}) \leq \Xi F$ using *IH* by *blast*

then show ?case

proof –

obtain *nn :: nat ⇒ nat ⇒ nat* where

$\forall x0\ x1. (\exists v2. x0 = x1 + v2) = (x0 = x1 + nn\ x0\ x1)$

by *moura*

then have $\Xi F = \Xi (A - \{a\}) + nn\ (\Xi F)\ (\Xi (A - \{a\}))$

using *Nat.le-iff-add* $\langle \Xi (A - \{a\}) \leq \Xi F \rangle$ 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*:

$\{\# La : \# A + \{\# L\#\}. 2 \leq \text{count } (A + \{\# L\#\})\ La\#\} = \{\# La : \# A. La \neq L \wedge 2 \leq \text{count } A\ La\#\}$

$\# \cup (\text{if count } A\ L \geq 1 \text{ then replicate-mset (count } A\ L + 1)\ L \text{ else } \{\#\})$

by (*auto intro: multiset-eqI*)

lemma *mset-condensation2*:

$\{\# La : \# A + \{\# L\#\} + \{\# L\#\}. 2 \leq \text{count } (A + \{\# L\#\} + \{\# L\#\})\ La\#\} = \{\# La : \# A. La \neq L \wedge$

$2 \leq \text{count } A\ La\#\} \# \cup (\text{replicate-mset (count } A\ L + 2)\ L)$

by (*auto intro: multiset-eqI*)

lemma *msetsum-disjoint*:

assumes $A \# \cap B = \{\#\}$

shows $(\sum La \in \# A \# \cup B. f\ La) =$

$(\sum La \in \# A. f\ La) + (\sum La \in \# B. f\ La)$

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 $C\ D :: 'a \Rightarrow 'b :: \{\text{comm-monoid-add}\}$

shows $(\sum x \in \# A. C\ x + D\ x) = (\sum x \in \# A. C\ x) + (\sum x \in \# A. D\ x)$

by (*induction A*) (*auto simp: ac-simps*)

lemma *msetsum-if-eq[simp]*: $(\sum x \in \#A. \text{if } L = x \text{ then } 1 \text{ else } 0) = \text{count } A \ L$
by (*induction A*) *auto*

lemma *filter-equality-in-mset*:
 $\text{filter-mset } (op = L) \ A = \text{replicate-mset } (\text{count } A \ L) \ L$
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.prems*)
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: insert (A + {#L#} + {#L#}) N = N
    using f3 f2 Un-empty-right condensation.hyps insert-iff by fastforce
  have  $\bigwedge m M.$  insert (m::'a 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  $\langle \Xi \{A + \{ \#L\# \} \} < \Xi \{A + \{ \#L\# \} + \{ \#L\# \} \rangle$ 
      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  $\langle \Xi \{A + \{ \#L\# \} \} < \Xi \{A + \{ \#L\# \} + \{ \#L\# \} \rangle$  condensation.hyps fin
    sum-count-ge-2.remove[of - A + {#L#} + {#L#}] (?C'  $\notin$  N)
  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  $\text{simplified } N \vee (\exists N'. \text{full1 simplify } N\ N')$ 
  using rtrancl-simplify-terminates[OF assms] unfolding full1-def
  by (metis Nitpick.rtrancl-unfold)

```

lemma *finite-simplified-full-simp*:

```

  assumes finite N
  shows  $\exists N'. \text{full simplify } N\ N'$ 
  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  $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{resolution}^{**}\ \psi\ \psi' \wedge \text{partial-interps tree}'\ I\ (\text{fst } \psi')$ 
     $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$ 
  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)

```

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  $\wedge$  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 χ χ' where

```
   $\chi$ :  $\neg I \cup \{Pos\ v\} \models \chi$  and
  tot $\chi$ : total-over-m ( $I \cup \{Pos\ v\}$ )  $\{\chi\}$  and
   $\chi\psi$ :  $\chi \in fst\ \psi$  and
   $\chi'$ :  $\neg I \cup \{Neg\ v\} \models \chi'$  and
  tot $\chi'$ : total-over-m ( $I \cup \{Neg\ v\}$ )  $\{\chi'\}$  and  $\chi'\psi$ :  $\chi' \in fst\ \psi$ 
  using part unfolding xs by auto
  have Posv: Pos v  $\notin \# \chi$  using  $\chi$  unfolding true-cls-def true-lit-def by auto
  have Negv: Neg v  $\notin \# \chi'$  using  $\chi'$  unfolding true-cls-def true-lit-def by auto
  {
    assume Neg $\chi$ :  $\neg Neg\ v \in \# \chi$ 
    then have  $\neg I \models \chi$  using  $\chi$  Posv unfolding true-cls-def true-lit-def by auto
    moreover have 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 resolution**  $\psi\ \psi$ 
      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 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 resolution**  $\psi\ \psi$  using  $\chi'\psi$  I $\chi$  unfolding xs by auto
```

}

moreover {

```
  assume neg: Neg v  $\in \# \chi$  and pos: Pos v  $\in \# \chi'$ 
  have count  $\chi$  (Neg v) = 1
    using simplified-count[OF simp  $\chi\psi$ ] neg by (metis One-nat-def Suc-le-mono Suc-pred eq-iff le0)
  have count  $\chi'$  (Pos v) = 1
    using simplified-count[OF simp  $\chi'\psi$ ] pos by (metis One-nat-def Suc-le-mono Suc-pred eq-iff le0)
  obtain C where  $\chi C$ :  $\chi = C + \{\#Neg\ v\# \}$  and negC: Neg v  $\notin \# C$  and posC: Pos v  $\notin \# C$ 
  proof -
    assume a1:  $\bigwedge C. \llbracket \chi = C + \{\#Neg\ v\# \}; Neg\ v \notin \# C; Pos\ v \notin \# C \rrbracket \implies thesis$ 
    have f2:  $\bigwedge n. (0::nat) + n = n$ 
```

```

    by simp
  obtain mm :: 'v literal multiset  $\Rightarrow$  'v literal  $\Rightarrow$  'v literal multiset where
    f3:  $\{\#Neg\ v\#\} + mm\ \chi\ (Neg\ v) = \chi$ 
    by (metis (no-types)  $\langle count\ \chi\ (Neg\ v) = 1 \rangle$  add.commute multi-member-split
        zero-less-one)
  then have Pos v  $\notin\#$  mm  $\chi\ (Neg\ v)$ 
    using f2 by (metis (no-types) Posv  $\langle count\ \chi\ (Neg\ v) = 1 \rangle$  add.right-neutral
        add-left-cancel count-single count-union less-nat-zero-code)
  then show ?thesis
    using f3 a1 by (metis (no-types)  $\langle count\ \chi\ (Neg\ v) = 1 \rangle$  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  $\langle count\ \chi' (Pos\ v) = 1 \rangle$  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) rtrancpl.rtrancpl-into-rtrancpl rtrancpl.rtrancpl-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  $\in\#$   $(\{\#Pos\ v\#\} + C') \wedge Neg\ p \in\# (\{\#Neg\ v\#\} + C)$ 
         $\wedge ((\exists \chi \in fst\ \psi. (\forall I. total-over-m\ I\ \{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + (\{\#Neg\ v\#\}$ 
         $+ C) - \{\#Neg\ p\#\})) \longrightarrow total-over-m\ I\ \{\chi\}) \wedge (\forall I. total-over-m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\#Pos$ 

```



```

 $v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})) \vee \text{tautology } ((\{\#Pos\ v\#\} + C') -$ 
 $\{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$ 
  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  $\exists \chi \in fst\ \psi. (\forall I. total-over-m\ I\ \{C+C'\} \longrightarrow total-over-m\ I\ \{\chi\})$ 
 $\wedge (\forall I. total-over-m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
  then obtain  $\vartheta$  where
     $\vartheta: \vartheta \in fst\ \psi$  and
     $tot\text{-}\vartheta\text{-}CC': \forall I. total-over-m\ I\ \{C+C'\} \longrightarrow total-over-m\ I\ \{\vartheta\}$  and
     $\vartheta\text{-}inv: \forall I. total-over-m\ I\ \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$  by blast
  have partial-interps Leaf I (fst  $\psi$ )
    using  $tot\text{-}\vartheta\text{-}CC'\ \vartheta\ \vartheta\text{-}inv\ totC\ totC' \hookrightarrow I \models C + C'$  total-over-m-sum by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case by blast
}
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  $(I \cup \{Pos\ v\})$  (fst  $\psi$ )
  and partad: partial-interps ad  $(I \cup \{Neg\ v\})$  (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover
    have sem-tree-size ag < sem-tree-size xs  $\implies finite\ (fst\ \psi) \implies already-used\text{-}inv\ \psi$ 
 $\implies partial\text{-}interps\ ag\ (I \cup \{Pos\ v\})\ (fst\ \psi) \implies simplified\ (fst\ \psi)$ 
 $\implies \exists tree'\ \psi'. resolution^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ (I \cup \{Pos\ v\})\ (fst\ \psi')$ 
 $\wedge (sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ ag \vee sem\text{-}tree\text{-}size\ ag = 0)$ 
    using IH[of ag I  $\cup \{Pos\ v\}$ ] by auto
  ultimately obtain  $\psi' :: 'v\ state$  and  $tree' :: 'v\ sem\text{-}tree$  where
    inf: resolution**  $\psi\ \psi'$ 
    and part: partial-interps tree'  $(I \cup \{Pos\ v\})$  (fst  $\psi'$ )

```

```

    and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
    using finite part rtranclp.rtrancl-refl a-u-i simp by blast

    have partial-interps ad (I ∪ {Neg v}) (fst ψ')
      using rtranclp-resolution-preserve-partial-tree inf partad by fast
    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) ⟶ simplified (fst ψ)
        ⟶ (∃ tree' ψ'. resolution** 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 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 ∨ 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: ¬satisfiable (fst ψ) and
    finite: finite (fst ψ) and
    a-u-v: already-used-inv ψ
  shows ∃ ψ'. (resolution** ψ ψ' ∧ {#} ∈ fst ψ')
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: ¬ {} ⊨ χ and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
    }
  }
end

```

```

    using H unfolding tree by auto
  moreover have  $\{\#\} = \chi$ 
    using H atms-empty-iff-empty tot $\chi$ 
    unfolding true-cls-def total-over-m-def total-over-set-def by fastforce
  moreover have resolution**  $\psi$   $\psi$  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  $\exists \psi'. \text{full1 simplify } (\text{fst } \psi) \psi'$ 
        by (metis Nitpick.rtranclp-unfold bigger.prem(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  $\langle \text{full1 simplify } (\text{fst } \psi) \text{ } N \rangle$  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.prem(1) rtranclp-resolution-preserve-partial-tree apply blast
  using  $\psi_0$  bigger.prem(2) rtranclp-resolution-preserves-unsat apply blast
  using  $\psi_0$  bigger.prem(3) rtranclp-resolution-finite apply blast
  using rtranclp-resolution-already-used-inv[OF  $\psi_0$  bigger.prem(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 rtranclp-resolution-preserves-already-used-inv:
  assumes resolution** S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
  apply (induct rule: rtranclp-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 rtranclp-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 rtranclp-preserves-sat tranclp-into-rtranclp unfolding full1-def apply fastforce
  by (metis fst-conv full-def inference-preserves-un-sat rtranclp-preserves-sat
    satisfiable-carac' satisfiable-def)

```

```

lemma rtranclp-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 rtrancpl-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  $H: \neg \text{thesis}$ 
  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 rtrancpl-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  $\neg$  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
      tranclp-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, 'wl, 'mark) marked-lit =
  is-marked: Marked (lit-of: 'v literal) (level-of: 'wl) |
  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*:
is-marked L $\implies \exists K \text{ lvl. } L = \text{Marked } K \text{ lvl}$
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-ms-lambda-lit-of-is-atm-of-lit-of[simp]*:
atms-of-ms (($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set M'*) = *atm-of* ' *lits-of M'*
unfolding *atms-of-ms-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-cl-def* **by** *simp*

lemma *true-annot-empty[simp]*:
 $\neg I \models_a \{\#\}$
unfolding *true-annot-def true-cl-def* **by** *simp*

lemma *empty-true-annot-def*[*iff*]:
 $\square \models_{as} \psi \longleftrightarrow \psi = \{\}$
unfolding *true-annot-def* **by** *auto*

lemma *true-annot-empty*[*simp*]:
 $I \models_{as} \{\}$
unfolding *true-annot-def* **by** *auto*

lemma *true-annot-single-true-annot*[*iff*]:
 $I \models_{as} \{C\} \longleftrightarrow I \models_a C$
unfolding *true-annot-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-annot-insert-l* [*simp*]:
 $M \models_{as} A \implies L \# M \models_{as} A$
unfolding *true-annot-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-annot-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-annot-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-annot-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-annot-def true-annot-def lits-of-def true-clss-def
true-clss-clss-def)

lemma true-annots-marked-true-clss[iff]:

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-clss *)

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

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]:

defined-lit (*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

$(\exists l. \text{Marked } (- \text{lit-of } x) \ l \in \text{set } I)$

$\vee (\exists l. \text{Marked } (\text{lit-of } x) \ l \in \text{set } I)$
 $\vee (\exists l. \text{Propagated } (\neg \text{lit-of } x) \ l \in \text{set } I)$
 $\vee (\exists l. \text{Propagated } (\text{lit-of } x) \ l \in \text{set } I)$
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*:

$\text{defined-lit } I \ L \longleftrightarrow ((\text{lits-of } I) \models L \vee (\text{lits-of } I) \models \neg L)$
unfolding *defined-lit-def* **by** (*auto simp add: lits-of-def rev-image-eqI*
dest!: literal-is-lit-of-marked)

lemma *consistent-interp* $(\text{lits-of } I) \Longrightarrow I \models_{\text{as}} N \Longrightarrow \text{satisfiable } N$
by (*simp add: true-annots-true-cl*)

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]*:

$\text{defined-lit } I \ (\neg L) \longleftrightarrow \text{defined-lit } I \ L$
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* :: $('v, 'l, 'm) \text{ marked-lits}$

$\Rightarrow ('v, 'l, 'm) \text{ marked-lits} \times ('v, 'l, 'm) \text{ marked-lits}$ **where**

backtrack-split $[] = ([], [])$ |

backtrack-split $(\text{Propagated } L \ P \ \# \ \text{mlits}) = \text{apfst } ((\text{op } \#) \ (\text{Propagated } L \ P)) \ (\text{backtrack-split } \text{mlits})$ |

backtrack-split $(\text{Marked } L \ l \ \# \ \text{mlits}) = ([], \text{Marked } L \ l \ \# \ \text{mlits})$

lemma *backtrack-split-fst-not-marked*: $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \Longrightarrow \neg \text{is-marked } a$
by (*induct l rule: marked-lit-list-induct*) *auto*

lemma *backtrack-split-snd-hd-marked*:

snd (backtrack-split l) ≠ [] ⇒ is-marked (hd (snd (backtrack-split l)))
by (induct l rule: marked-lit-list-induct) auto

lemma *backtrack-split-list-eq[simp]*:
fst (backtrack-split l) @ (snd (backtrack-split l)) = l
by (induct l rule: marked-lit-list-induct) auto

lemma *backtrack-snd-empty-not-marked*:
backtrack-split M = (M'', []) ⇒ ∀ l ∈ set M. ¬ 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*:
 $\exists l \in \text{set } M. \text{is-marked } l \Rightarrow \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$
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*:
backtrack-split M = (takeWhile (Not o is-marked) M, dropWhile (Not o 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* [] = [([], [])] is necessary otherwise, we can call the *hd* function in the other pattern.

fun *get-all-marked-decomposition* :: ('a, 'l, 'm) marked-lits
 $\Rightarrow ((('a, 'l, 'm) \text{ marked-lits} \times ('a, 'l, 'm) \text{ marked-lits}) \text{ list}) \text{ where}$
get-all-marked-decomposition (Marked L l # Ls) =
 (Marked L l # Ls, []) # *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 [] = [([], [])]

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 = [] ⇔ False
by (induct M, simp) (case-tac a, auto)

lemma *get-all-marked-decomposition-never-empty-sym[iff]*:
 $[] = \text{get-all-marked-decomposition } M \iff \text{False}$
using *get-all-marked-decomposition-never-empty[of M]* **by** presburger

lemma *get-all-marked-decomposition-decomp*:
 $\text{hd} (\text{get-all-marked-decomposition } S) = (a, c) \Rightarrow S = c @ a$
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:
  backtrack-split S = (M, M')  $\longleftrightarrow$  hd (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) ∈ 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[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  $\text{length } (\text{get-all-marked-decomposition } (M' @ M''))$ 
    =  $\text{length } (\text{get-all-marked-decomposition } M'')$ 
  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  $\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)) \neq []$ 
  and  $\forall l \in \text{set } M'. \neg \text{is-marked } l$ 
  and  $\text{hd } (\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M))) = (M0', M0)$ 
  shows  $\text{hd } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$ 
  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  $\text{tl } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$ 
    =  $\text{tl } (\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)))$ 
  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  $\text{tl } M = M0' @ M0 \wedge \text{is-marked } (\text{hd } M)$ 
  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 = this(1) and g = this(2)
  {
    fix L level
    assume a:  $a = \text{Marked } L \text{ level}$ 
    have  $Ls = M0' @ M0$ 
    using g a by (force intro: get-all-marked-decomposition-decomp)
    hence  $\text{tl } M = M0' @ M0 \wedge \text{is-marked } (\text{hd } M)$  using g a by auto
  }
  moreover {
    fix L P
    assume a:  $a = \text{Propagated } L P$ 
    have  $\text{tl } M = M0' @ M0 \wedge \text{is-marked } (\text{hd } M)$ 
    using IH Cons.premis unfolding a by (cases get-all-marked-decomposition Ls) 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, 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, 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 (get-all-marked-decomposition M)
shows  $N \cup \{\{\#lit\text{-of } L\# \mid L. \text{is-marked } L \wedge L \in \text{set } M\}\}$ 
 $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (\text{set ' snd ' set (get-all-marked-decomposition M))$ 
using assms
proof (induct length (get-all-marked-decomposition M) arbitrary: M)
  case 0
  thus ?case by auto
next
case (Suc n) note IH = this(1) and length = 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.prem1 g by auto
  }
  moreover {
    assume l: length a = 1 and m: is-marked (hd a) and hd: hd a ∈ set M
    hence  $(\lambda a. \{\#lit\text{-of } a\# \}) (\text{hd } a) \in \{\{\#lit\text{-of } L\# \mid L. \text{is-marked } L \wedge L \in \text{set } M\}\}$  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 ⊆ N ∪ {#lit-of m#} ' set b
    using Suc.prem1 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.prem1 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.prem1 unfolding Ls0 all-decomposition-implies-def by auto
  hence N: N ∪ {#lit-of L#} | L. is-marked L ∧ L ∈ set M'
     $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (\text{set ' snd ' set (get-all-marked-decomposition M')})$ 
    using IH length' by auto
}
}

```



```

have l:  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M'\}$ 
   $\subseteq N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
  using  $M'\text{-in-}M$  by auto
hence  $\Psi N: N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
   $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (\text{set ' snd ' set (get-all-marked-decomposition } M'))$ 
  using  $\text{true-clss-clss-subset}[OF \ l \ N]$  by auto
have  $\text{is-marked (hd } Ls0)$  and  $LS: tl \ Ls0 = \text{seen1 } @ \ Ls1$ 
  using  $\text{get-all-marked-decomposition-hd-hd}[of \ M]$  unfolding  $Ls0 \ Ls1$  by auto

have  $LSM: \text{seen1 } @ \ Ls1 = M'$  using  $\text{get-all-marked-decomposition-decomp}[of \ M'] \ Ls1$  by auto
have  $M': \text{set } M' = \text{Union (set ' snd ' set (get-all-marked-decomposition } M'))$ 
   $\cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M'\}$ 
  using  $\text{get-all-marked-decomposition-snd-union}$  by auto

{
  assume  $Ls0 \neq []$ 
  hence  $hd \ Ls0 \in \text{set } M$  using  $\text{get-all-marked-decomposition-fst-empty-or-hd-in-}M \ Ls0$  by blast
  hence  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} \models_p (\lambda a. \{\#lit\text{-of } a\# \}) (hd \ Ls0)$ 
    using  $\langle \text{is-marked (hd } Ls0) \rangle$  by  $(metis \ (\text{mono-tags, lifting}) \ UnCI \ \text{mem-Collect-eq} \ \text{true-clss-clss-in})$ 
} note  $hd\text{-}Ls0 = \text{this}$ 

have l:  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } (\bigcup (\text{set ' snd ' set (get-all-marked-decomposition } M'))$ 
   $\cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M'\})$ 
   $= (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' }$ 
   $\bigcup (\text{set ' snd ' set (get-all-marked-decomposition } M'))$ 
   $\cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M'\}$ 
  by auto
have  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M'\} \models_{ps}$ 
   $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } (\bigcup (\text{set ' snd ' set (get-all-marked-decomposition } M'))$ 
   $\cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M'\})$ 
  unfolding  $l$  using  $N$  by  $(\text{auto simp add: all-in-true-clss-clss})$ 
hence  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M'\} \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set (tl } Ls0)$ 
  using  $M'$  unfolding  $LS \ LSM$  by auto
hence  $t: N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M'\}$ 
   $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set (tl } Ls0)$ 
  by  $(\text{blast intro: all-in-true-clss-clss})$ 
hence  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
   $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set (tl } Ls0)$ 
  using  $M'\text{-in-}M \ \text{true-clss-clss-subset}[OF \ t,$ 
     $\text{of } N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M'\}]$ 
  by auto
hence  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } Ls0$ 
  using  $hd\text{-}Ls0$  by  $(\text{case-tac } Ls0, \text{ auto})$ 

moreover have  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } Ls0 \cup N \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set seen0}$ 
  using  $\text{Suc.premis unfolding } Ls0 \ \text{all-decomposition-implies-def}$  by simp
moreover have  $\bigwedge M \ Ma. (M::'a \ \text{literal multiset set}) \cup Ma \models_{ps} M$ 
  by  $(\text{simp add: all-in-true-clss-clss})$ 
ultimately have  $\Psi: N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} \models_{ps}$ 
   $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set seen0}$ 
  by  $(\text{meson true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r})$ 
have  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' (set seen0}$ 
   $\cup (\bigcup_{x \in \text{set (get-all-marked-decomposition } M')} . \text{set (snd } x)))$ 
   $= (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set seen0}$ 

```

$\cup (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } (\bigcup_{x \in \text{set}} (\text{get-all-marked-decomposition } M'). \text{ set } (\text{snd } x))$
by *auto*

hence *?case* **unfolding** *Ls0* **using** $\Psi \Psi N$ **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 $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{ is-marked } L \wedge L \in \text{set } M\} \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } M$
(is ?I \models_{ps} ?A)

proof –

have *?I* $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \{L \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}$
by (*auto intro: all-in-true-clss-clss*)

moreover have *?I* $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (\text{set ' snd ' set } (\text{get-all-marked-decomposition } M))$
using *all-decomposition-implies-trail-is-implied* *assms* **by** *blast*

ultimately have $N \cup \{\{\#lit\text{-of } m\# \} \mid m. \text{ is-marked } m \wedge m \in \text{set } M\} \models_{ps} (\lambda m. \{\#lit\text{-of } m\# \}) \text{ ' } \bigcup (\text{set ' snd ' set } (\text{get-all-marked-decomposition } M))$
 $\cup (\lambda m. \{\#lit\text{-of } m\# \}) \text{ ' } \{m \mid m. \text{ is-marked } m \wedge m \in \text{set } M\}$
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 $\{\#L\# \} \in \text{CNot } \psi \longleftrightarrow -L \in \# \psi$
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-imply-uminus*:
assumes $L \in \# D$
and $M \models_{as} \text{CNot } D$
shows $M \models_a \{\#-L\# \}$ **and** $-L \in \text{lits-of } M$
using *assms* **by** (*auto simp add: true-annot-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-ms-CNot-atms-of[simp]*:
 $atms-of-ms\ (CNot\ C) = atms-of\ C$
unfolding *atms-of-ms-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-ms-CNot-atms-of atms-of-ms-insert atms-of-ms-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 $atm-of\ L \in atm-of\ \text{'lits-of } M$
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 \cup CNot {#L#}) **and**
cons: consistent-interp I **and**
 $I \models_s B$
have *total-over-m I ({#L#} \cup 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*:

$M \models_{as} CNot\ C \longleftrightarrow (\forall L \in \# C. -L \in lits\ of\ M)$

unfolding *CNot-def true-annots-true-cls true-clss-def* **by** *auto*

lemma *consistent-CNot-not-tautology*:

consistent-interp $M \implies M \models_s CNot\ D \implies \neg tautology\ D$

by (*metis atms-of-ms-CNot-atms-of consistent-CNot-not satisfiable-carac' satisfiable-def tautology-def total-over-m-def*)

lemma *atms-of-ms-CNot-atms-of-ms*: $atms-of-ms\ (CNot\ CC) = atms-of-ms\ \{CC\}$

by *simp*

lemma *total-over-m-CNot-toal-over-m[simp]*:

total-over-m $I\ (CNot\ C) = total-over-set\ I\ (atms-of\ C)$

unfolding *total-over-m-def total-over-set-def* **by** *auto*

lemma *uminus-lit-swap*: $\neg(a::'a\ literal) = i \longleftrightarrow a = -i$

by *auto*

lemma *true-clss-cls-plus-CNot*:

assumes *CC-L*: $A \models_p CC + \{\#L\# \}$

and *CNot-CC*: $A \models_{ps} CNot\ CC$

shows $A \models_p \{\#L\# \}$

unfolding *true-clss-clss-def true-clss-cls-def CNot-def total-over-m-def*

proof (*intro allI impI*)

fix I

assume *tot*: *total-over-set* $I\ (atms-of-ms\ (A \cup \{\{\#L\#\}\}))$

and *cons*: *consistent-interp* I

and $I: I \models_s A$

let $?I = I \cup \{Pos\ P | P. P \in atms-of\ CC \wedge P \notin atm-of\ 'I\}$

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*: *total-over-m* $?I\ (A \cup CNot\ CC)$

using *tot atms-of-s-def* **by** (*fastforce simp add: total-over-m-def total-over-set-def*)

hence *tot-I-A-CC-L*: *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-cls-def* **by** *blast*

moreover

have $?I \models_s CNot\ CC$ **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-ms-CNot-atms-of-ms atms-of-ms-union cons' consistent-CNot-not tot-CNot total-over-m-def true-clss-cls-def*)

hence $\neg ?I \models CC$ **using** $\langle ?I \models_s CNot\ CC \rangle cons'$ *consistent-CNot-not* **by** *blast*

ultimately have $?I \models \{\#L\# \}$ **by** *blast*

thus $I \models \{\#L\# \}$

by (*metis (no-types, lifting) atms-of-ms-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*: $L \# M \models_{as} CNot\ A$ **and** *LA*: $lit-of\ L \notin \# A - lit-of\ L \notin \# A$

shows $M \models_{as} CNot\ A$
using *LM unfolding true-annots-def Ball-def*
proof (*intro allI impI*)
fix l
assume $H: \forall x. x \in CNot\ A \longrightarrow L \# M \models_a x$ **and** $l: l \in CNot\ A$
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-of (lit-of a) \notin atms-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 $\forall x \in \text{atms-of } D. x \notin \text{atm-of ' lits-of } M$
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 $\forall x \in \text{atms-of-ms } D. x \notin \text{atm-of ' lits-of } M$
shows $M' \models_{as} D$ **unfolding** *true-annots-def*
using *assms true-annot-remove-if-notin-vars[of M M']*
unfolding *true-annots-def atms-of-ms-def* **by** *force*

lemma *all-variables-defined-not-imply-cnot*:
assumes $\forall s \in \text{atms-of-ms } \{B\}. s \in \text{atm-of ' lits-of } A$
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** $\neg \text{lits-of } A \models_l - L$
hence $\text{atm-of } L \in \text{atm-of ' lits-of } A$
using *assms(1) by (simp add: atm-of-lit-in-atms-of lits-of-def)*
hence $L \in \text{lits-of } A \vee -L \in \text{lits-of } A$
using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* **by** *metis*
hence $L \in \text{lits-of } A$ **using** $\langle \neg \text{lits-of } A \models_l - L \rangle$ **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 $\text{no-dup } L \equiv \text{distinct } (\text{map } (\lambda l. \text{atm-of } (\text{lit-of } l)) L)$

lemma $\text{no-dup-rev}[simp]$:

$\text{no-dup } (\text{rev } M) \longleftrightarrow \text{no-dup } M$

by $(\text{auto simp: rev-map}[symmetric])$

lemma $\text{no-dup-length-eq-card-atm-of-lits-of}$:

assumes $\text{no-dup } M$

shows $\text{length } M = \text{card } (\text{atm-of } ' \text{lits-of } M)$

using $\text{assms unfolding lits-of-def by (induct } M) (\text{auto simp add: image-image})$

lemma $\text{distinctconsistent-interp}$:

$\text{no-dup } M \implies \text{consistent-interp } (\text{lits-of } M)$

proof $(\text{induct } M)$

case Nil

show $?case$ **by** auto

next

case $(\text{Cons } L M)$

hence $a1$: $\text{consistent-interp } (\text{lits-of } M)$ **by** auto

have $a2$: $\text{atm-of } (\text{lit-of } L) \notin (\lambda l. \text{atm-of } (\text{lit-of } l)) ' \text{set } M$ **using** $\text{Cons.premis by auto}$

have $\text{undefined-lit } M (\text{lit-of } L)$

using $a2 \text{ image-iff unfolding defined-lit-def by fastforce}$

thus $?case$

using $a1$ **by** simp

qed

lemma $\text{distinct-get-all-marked-decomposition-no-dup}$:

assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$

and $\text{no-dup } M$

shows $\text{no-dup } (a @ b)$

using assms by force

lemma $\text{true-annots-lit-of-notin-skip}$:

assumes $L \# M \models_{\text{as}} \text{CNot } A$

and $\neg \text{lit-of } L \notin \# A$

and $\text{no-dup } (L \# M)$

shows $M \models_{\text{as}} \text{CNot } A$

proof $-$

have $\forall l \in \# A. \neg l \in \text{lits-of } (L \# M)$

using $\text{assms}(1) \text{ in-CNot-implies-uminus}(2) \text{ by blast}$

moreover

have $\text{atm-of } (\text{lit-of } L) \notin \text{atm-of } ' \text{lits-of } M$

using $\text{assms}(3) \text{ unfolding lits-of-def by force}$

hence $\neg \text{lit-of } L \notin \text{lits-of } M$ **unfolding** lits-of-def

by $(\text{metis (no-types) atm-of-uminus imageI})$

ultimately have $\forall l \in \# A. \neg l \in \text{lits-of } M$

using $\text{assms}(2) \text{ unfolding Ball-mset-def by (metis insertE lits-of-cons uminus-of-uminus-id)}$

thus $?thesis$ **by** $(\text{auto simp add: true-annots-def})$

qed

type-synonym $'v \text{ clauses} = 'v \text{ clause multiset}$

abbreviation $\text{true-annots-mset (infix } \models_{\text{asm}} 50) \text{ where}$

$I \models_{\text{asm}} C \equiv I \models_{\text{as}} (\text{set-mset } C)$

abbreviation *true-clss-clss-m*:: 'a clauses \Rightarrow 'a clauses \Rightarrow bool (**infix** \models_{psm} 50) **where**
 $I \models_{psm} C \equiv \text{set-mset } I \models_{ps} (\text{set-mset } C)$

Analog of $\llbracket ?N \models_{ps} ?B; ?A \subseteq ?B \rrbracket \Longrightarrow ?N \models_{ps} ?A$

lemma *true-clss-clssm-subsetE*: $N \models_{psm} B \Longrightarrow A \subseteq\# B \Longrightarrow N \models_{psm} A$
using *set-mset-mono true-clss-clss-subsetE* **by** *blast*

abbreviation *true-clss-clss-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-msu* **where**
 $\text{atms-of-msu } U \equiv \text{atms-of-ms } (\text{set-mset } U)$

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-ms-single-atm-of*[*simp*]:
 $\text{atms-of-ms } \{\{\# \text{lit-of } L\# \} \mid L. P L\} = \text{atm-of } \{ \text{lit-of } L \mid L. P L \}$
unfolding *atms-of-ms-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-ms-single-image-atm-of-lit-of*:

atms-of-ms $((\lambda x. \{\#lit\text{-of } x\}) \text{ ' } A) = atm\text{-of ' } (lit\text{-of ' } A)$
unfolding *atms-of-ms-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 $\mu_C :: nat \Rightarrow nat \Rightarrow nat \text{ list} \Rightarrow nat$ **where**
 $\mu_C \text{ s b M} \equiv (\sum i=0..<length \text{ M}. M!i * b^\wedge (s + i - length \text{ M}))$

lemma $\mu_C\text{-nil}[simp]$:
 $\mu_C \text{ s b []} = 0$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma $\mu_C\text{-single}[simp]$:
 $\mu_C \text{ s b [L]} = L * b^\wedge (s - Suc \ 0)$
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*:
 $(\sum i=1..<Suc \ j. f \ i) = (\sum i=0..<j. f \ (Suc \ i))$
using *set-sum-atLeastLessThan-add[of - 1 j]* **by** *force*

lemma $\mu_C\text{-cons}$:
 $\mu_C \text{ s b (L \# M)} = L * b^\wedge (s - 1 - length \text{ M}) + \mu_C \text{ s b M}$
proof –
have $\mu_C \text{ s b (L \# M)} = (\sum i=0..<length \text{ (L\#M)}. (L\#M)!i * b^\wedge (s + i - length \text{ (L\#M)}))$
unfolding $\mu_C\text{-def}$ **by** *blast*
also have $\dots = (\sum i=0..<1. (L\#M)!i * b^\wedge (s + i - length \text{ (L\#M)}))$
 $+ (\sum i=1..<length \text{ (L\#M)}. (L\#M)!i * b^\wedge (s + i - length \text{ (L\#M)}))$
by (*rule setsum-add-nat-ivl[symmetric]*) *simp-all*
finally have $\mu_C \text{ s b (L \# M)} = L * b^\wedge (s - 1 - length \text{ M})$
 $+ (\sum i=1..<length \text{ (L\#M)}. (L\#M)!i * b^\wedge (s + i - length \text{ (L\#M)}))$
by *auto*
moreover {
have $(\sum i=1..<length \text{ (L\#M)}. (L\#M)!i * b^\wedge (s + i - length \text{ (L\#M)})) =$
 $(\sum i=0..<length \text{ (M)}. (L\#M)!(Suc \ i) * b^\wedge (s + (Suc \ i) - length \text{ (L\#M)}))$
unfolding *length-Cons set-sum-atLeastLessThan-Suc* **by** *blast*
also have $\dots = (\sum i=0..<length \text{ (M)}. M!i * b^\wedge (s + i - length \text{ M}))$
by *auto*
finally have $(\sum i=1..<length \text{ (L\#M)}. (L\#M)!i * b^\wedge (s + i - length \text{ (L\#M)})) = \mu_C \text{ s b M}$
unfolding $\mu_C\text{-def}$.
}
ultimately show *?thesis* **by** *presburger*
qed

lemma $\mu_C\text{-append}$:
assumes $s \geq length \text{ (M@M')}$
shows $\mu_C \text{ s b (M@M')} = \mu_C (s - length \text{ M'}) \text{ b M} + \mu_C \text{ s b M'}$
proof –
have $\mu_C \text{ s b (M@M')} = (\sum i=0..<length \text{ (M@M')}. (M@M')!i * b^\wedge (s + i - length \text{ (M@M')}))$
unfolding $\mu_C\text{-def}$ **by** *blast*
moreover then have $\dots = (\sum i=0..<length \text{ M}. (M@M')!i * b^\wedge (s + i - length \text{ (M@M')}))$

$+$ $(\sum i=length\ M..<length\ (M@M')). (M@M')!i * b^\wedge (s+i - length\ (M@M')))$
by *(auto intro!: setsum-add-nat-ivl[symmetric])*
moreover
have $\forall i \in \{0..<length\ M\}. (M@M')!i * b^\wedge (s+i - length\ (M@M')) = M!i * b^\wedge (s - length\ M' + i - length\ M)$
using $\langle s \geq length\ (M@M') \rangle$ **by** *(auto simp add: nth-append ac-simps)*
then have $\mu_C\ (s - length\ M')\ b\ M = (\sum i=0..<length\ M. (M@M')!i * b^\wedge (s+i - length\ (M@M')))$
unfolding μ_C -def **by** *auto*
ultimately have $\mu_C\ s\ b\ (M@M') = \mu_C\ (s - length\ M')\ b\ M$
 $+$ $(\sum i=length\ M..<length\ (M@M')). (M@M')!i * b^\wedge (s+i - length\ (M@M')))$
by *auto*
moreover {
have $(\sum i=length\ M..<length\ (M@M')). (M@M')!i * b^\wedge (s+i - length\ (M@M')) =$
 $(\sum i=0..<length\ M'. M!i * b^\wedge (s+i - length\ M'))$
unfolding *length-append set-sum-atLeastLessThan-add* **by** *auto*
then have $(\sum i=length\ M..<length\ (M@M')). (M@M')!i * b^\wedge (s+i - length\ (M@M')) = \mu_C\ s\ b\ M'$
unfolding μ_C -def .
}
ultimately show *?thesis* **by** *presburger*
qed

lemma μ_C -cons-non-empty-inf:
assumes *M-ge-1*: $\forall i \in set\ M. i \geq 1$ **and** *M*: $M \neq []$
shows $\mu_C\ s\ b\ M \geq b^\wedge (s - length\ M)$
using *assms* **by** *(cases M) (auto simp: mult-eq-if μ_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 μ_C -bounded-non-degenerated:
fixes $b :: nat$
assumes
 $b > 0$ **and**
 $M \neq []$ **and**
 M -le: $\forall i < length\ M. M!i < b$ **and**
 $s \geq length\ M$
shows $\mu_C\ s\ b\ M < b^\wedge s$

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 $\forall i < length\ M. M!i = 0$ **using** *M-le* **by** *auto*
then have $\mu_C\ s\ b\ M = 0$ **unfolding** μ_C -def **by** *auto*
then show *?thesis* **using** $\langle b > 0 \rangle$ **by** *auto*
next
case *b*
have $\forall i \in \{0..<length\ M\}. M!i * b^\wedge (s+i - length\ M) \leq (b-1) * b^\wedge (s+i - length\ M)$

```

    using M-le ⟨b > 1⟩ by auto
  then have  $\mu_C s b M \leq (\sum i=0..<length\ M. (b-1) * b^\wedge (s+i - length\ M))$ 
    using ⟨M≠[]⟩ ⟨b>0⟩ unfolding  $\mu_C$ -def by (auto intro: setsum-mono)
  also
    have  $\forall i \in \{0..<length\ M\}. (b-1) * b^\wedge (s+i - length\ M) = (b-1) * b^\wedge i * b^\wedge (s - length\ M)$ 
      by (metis Nat.add-diff-assoc2 add.commute assms(4) mult.assoc power-add)
    then have  $(\sum i=0..<length\ M. (b-1) * b^\wedge (s+i - length\ M))$ 
      =  $(\sum i=0..<length\ M. (b-1) * b^\wedge i * b^\wedge (s - length\ M))$ 
      by (auto simp add: ac-simps)
    also have  $\dots = (\sum i=0..<length\ M. b^\wedge i) * b^\wedge (s - length\ M) * (b-1)$ 
      by (simp add: setsum-left-distrib setsum-right-distrib ac-simps)
    finally have  $\mu_C s b M \leq (\sum i=0..<length\ M. b^\wedge i) * (b-1) * b^\wedge (s - length\ M)$ 
      by (simp add: ac-simps)

  also
    have  $(\sum i=0..<length\ M. b^\wedge i) * (b-1) = b^\wedge (length\ M) - 1$ 
      using sum-of-powers[of b length M] ⟨b>1⟩
      by (auto simp add: ac-simps)
    finally have  $\mu_C s b M \leq (b^\wedge (length\ M) - 1) * b^\wedge (s - length\ M)$ 
      by auto
    also have  $\dots < b^\wedge (length\ M) * b^\wedge (s - length\ M)$ 
      using ⟨b>1⟩ by auto
    also have  $\dots = b^\wedge s$ 
      by (metis assms(4) le-add-diff-inverse power-add)
    finally show ?thesis unfolding  $\mu_C$ -def by (auto simp add: ac-simps)
qed

```

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

```

lemma  $\mu_C$ -bounded:
  fixes b :: nat
  assumes
    M-le:  $\forall i < length\ M. M!i < b$  and
    s  $\geq length\ M$ 
    b > 0
  shows  $\mu_C s b M < b^\wedge s$ 
proof -
  consider (M0)  $M = [] \mid (M) b > 0$  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 ⟨b > 0⟩ by auto
    next
      case M
        show ?thesis using  $\mu_C$ -bounded-non-degenerated[OF M assms(1,2)] by arith
  qed
qed

```

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

```

lemma  $\mu_C$ -base-0:
  assumes length M  $\leq s$ 
  shows  $\mu_C s 0 M \leq M!0$ 
proof -

```

```

{
  assume  $s = \text{length } M$ 
  moreover {
    fix  $n$ 
    have  $(\sum_{i=0..<n}. M ! i * (0::\text{nat}) ^ i) \leq M ! 0$ 
    apply (induction  $n$  rule: nat-induct)
    by simp (case-tac  $n$ , auto)
  }
  ultimately have ?thesis unfolding  $\mu_C$ -def by auto
}
moreover
{
  assume  $\text{length } M < s$ 
  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-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
  assumes
    trail-prepend-trail[simp]:
       $\bigwedge st L. \text{undefined-lit } (\text{trail } st) (\text{lit-of } L) \Longrightarrow \text{trail } (\text{prepend-trail } L \text{ } st) = L \# \text{trail } st$ 
    and
    tl-trail[simp]: trail (tl-trail  $S$ ) = tl (trail  $S$ ) and
    trail-add-clNOT[simp]:  $\bigwedge st C. \text{no-dup } (\text{trail } st) \Longrightarrow \text{trail } (\text{add-cl}_{NOT} C \text{ } st) = \text{trail } st$  and
    trail-remove-clNOT[simp]:  $\bigwedge st C. \text{trail } (\text{remove-cl}_{NOT} C \text{ } st) = \text{trail } st$  and

    clauses-prepend-trail[simp]:
       $\bigwedge st L. \text{undefined-lit } (\text{trail } st) (\text{lit-of } L) \Longrightarrow \text{clauses } (\text{prepend-trail } L \text{ } st) = \text{clauses } st$ 
    and
    clauses-tl-trail[simp]:  $\bigwedge st. \text{clauses } (\text{tl-trail } st) = \text{clauses } st$  and
    clauses-add-clNOT[simp]:
       $\bigwedge st C. \text{no-dup } (\text{trail } st) \Longrightarrow \text{clauses } (\text{add-cl}_{NOT} C \text{ } st) = \{\#C\} + \text{clauses } st$  and
    clauses-remove-clNOT[simp]:  $\bigwedge st C. \text{clauses } (\text{remove-cl}_{NOT} C \text{ } st) = \text{remove-mset } C (\text{clauses } st)$ 
  begin

  function reduce-trail-toNOT :: 'a list  $\Rightarrow$  'st  $\Rightarrow$  'st where
    reduce-trail-toNOT  $F S$  =
      (if length (trail  $S$ ) = length  $F \vee \text{trail } S = []$  then  $S$  else reduce-trail-toNOT  $F$  (tl-trail  $S$ ))
  by fast+
  termination by (relation measure ( $\lambda(-, S). \text{length } (\text{trail } S)$ )) auto
  declare reduce-trail-toNOT.simps[simp del]

  lemma

```

shows

reduce-trail-to_{NOT}-nil[simp]: $\text{trail } S = [] \implies \text{reduce-trail-to}_{NOT} F S = S$ **and**
reduce-trail-to_{NOT}-eq-length[simp]: $\text{length } (\text{trail } S) = \text{length } F \implies \text{reduce-trail-to}_{NOT} F S = S$
by (auto simp: *reduce-trail-to_{NOT}.simps*)

lemma *reduce-trail-to_{NOT}-length-ne*[simp]:

$\text{length } (\text{trail } S) \neq \text{length } F \implies \text{trail } S \neq [] \implies$
 $\text{reduce-trail-to}_{NOT} F S = \text{reduce-trail-to}_{NOT} F (\text{tl-trail } S)$
by (auto simp: *reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-length-le*:

assumes $\text{length } F > \text{length } (\text{trail } S)$
shows $\text{trail } (\text{reduce-trail-to}_{NOT} F S) = []$
using *assms* **by** (induction $F S$ rule: *reduce-trail-to_{NOT}.induct*)
(simp add: *less-imp-diff-less reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-nil*[simp]:

$\text{trail } (\text{reduce-trail-to}_{NOT} [] S) = []$
by (induction $[] S$ rule: *reduce-trail-to_{NOT}.induct*)
(simp add: *less-imp-diff-less reduce-trail-to_{NOT}.simps*)

lemma *clauses-reduce-trail-to_{NOT}-nil*:

$\text{clauses } (\text{reduce-trail-to}_{NOT} [] S) = \text{clauses } S$
by (induction $[] S$ rule: *reduce-trail-to_{NOT}.induct*)
(simp add: *less-imp-diff-less reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-drop*:

$\text{trail } (\text{reduce-trail-to}_{NOT} F S) =$
 (if $\text{length } (\text{trail } S) \geq \text{length } F$
 then $\text{drop } (\text{length } (\text{trail } S) - \text{length } F) (\text{trail } S)$
 else $[]$)
apply (induction $F S$ rule: *reduce-trail-to_{NOT}.induct*)
apply (rename-tac $F S$)
apply (case-tac $\text{trail } S$)
apply auto[]
apply (rename-tac list)
apply (case-tac $\text{Suc } (\text{length } \text{list}) > \text{length } F$)
 prefer 2 **apply** simp
apply (subgoal-tac $\text{Suc } (\text{length } \text{list}) - \text{length } F = \text{Suc } (\text{length } \text{list} - \text{length } F)$)
 apply simp
apply simp
done

lemma *reduce-trail-to_{NOT}-skip-beginning*:

assumes $\text{trail } S = F' @ F$
shows $\text{trail } (\text{reduce-trail-to}_{NOT} F S) = F$
using *assms* **by** (auto simp: *trail-reduce-trail-to_{NOT}-drop*)

lemma *reduce-trail-to_{NOT}-clauses*[simp]:

$\text{clauses } (\text{reduce-trail-to}_{NOT} F S) = \text{clauses } S$
by (induction $F S$ rule: *reduce-trail-to_{NOT}.induct*)
(simp add: *less-imp-diff-less reduce-trail-to_{NOT}.simps*)

abbreviation *trail-weight where*

trail-weight $S \equiv \text{map } ((\lambda l. 1 + \text{length } l) \circ \text{snd}) (\text{get-all-marked-decomposition } (\text{trail } S))$

definition *state-eq_{NOT}* :: 'st \Rightarrow 'st \Rightarrow bool (**infix** \sim 50) **where**
 $S \sim T \iff \text{trail } S = \text{trail } T \wedge \text{clauses } S = \text{clauses } T$

lemma *state-eq_{NOT}-ref[simp]*:
 $S \sim S$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma *state-eq_{NOT}-sym*:
 $S \sim T \iff T \sim S$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma *state-eq_{NOT}-trans*:
 $S \sim T \implies T \sim U \implies S \sim U$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma
shows
state-eq_{NOT}-trail: $S \sim T \implies \text{trail } S = \text{trail } T$ **and**
state-eq_{NOT}-clauses: $S \sim T \implies \text{clauses } S = \text{clauses } T$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemmas *state-simp_{NOT}[simp]* = *state-eq_{NOT}-trail* *state-eq_{NOT}-clauses*

lemma *trail-eq-reduce-trail-to_{NOT}-eq*:
 $\text{trail } S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F T)$
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 $\text{reduce-trail-to}_{\text{NOT}} F S \sim \text{reduce-trail-to}_{\text{NOT}} F T$
proof –
have $\text{clauses } (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{clauses } (\text{reduce-trail-to}_{\text{NOT}} F T)$
using ST **by** *auto*
moreover have $\text{trail } (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F T)$
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-cl_{NOT}[simp]*:
 $\text{no-dup } (\text{trail } S) \implies$
 $\text{trail } (\text{reduce-trail-to}_{\text{NOT}} F (\text{add-cl}_{\text{NOT}} C S)) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F S)$
by (*rule* *trail-eq-reduce-trail-to_{NOT}-eq*) *simp*

lemma *reduce-trail-to_{NOT}-trail-tl-trail-decomp[simp]*:
 $\text{trail } S = F' @ \text{Marked } K () \# F \implies$
 $\text{trail } (\text{reduce-trail-to}_{\text{NOT}} F (\text{tl-trail } S)) = F$
apply (*rule* *reduce-trail-to_{NOT}-skip-beginning*[*of* - $\text{tl } (F' @ \text{Marked } K () \# [])$])
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-cl_{NOT}* *remove-cl_{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-cl_{NOT} *remove-cl_{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]: $C + \{\#L\} \in \# \text{ clauses } S \implies \text{trail } S \models_{as} C \text{Not } C$
 $\implies \text{undefined-lit } (\text{trail } S) \text{ } L$
 $\implies \text{propagate-cond } (\text{Propagated } L \text{ } ()) \text{ } S$
 $\implies T \sim \text{prepend-trail } (\text{Propagated } L \text{ } ()) \text{ } S$
 $\implies \text{propagate}_{NOT} \text{ } S \text{ } T$
inductive-cases *propagateE*[elim]: *propagate_{NOT}* *S* *T*
end

locale *decide-ops* =
dpll-state *trail* *clauses* *prepend-trail* *tl-trail* *add-cl_{NOT}* *remove-cl_{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-cl_{NOT} *remove-cl_{NOT}* :: 'v clause \Rightarrow 'st \Rightarrow 'st
begin
inductive *decide_{NOT}* :: 'st \Rightarrow 'st \Rightarrow bool **where**
decide_{NOT}[intro]: $\text{undefined-lit } (\text{trail } S) \text{ } L \implies \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S)$
 $\implies T \sim \text{prepend-trail } (\text{Marked } L \text{ } ()) \text{ } S$
 $\implies \text{decide}_{NOT} \text{ } S \text{ } T$
inductive-cases *decideE*[elim]: *decide_{NOT}* *S* *S'*
end

locale *backjumping-ops* =
dpll-state *trail* *clauses* *prepend-trail* *tl-trail* *add-cl_{NOT}* *remove-cl_{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-cl_{NOT} *remove-cl_{NOT}* :: 'v clause \Rightarrow 'st \Rightarrow 'st +
fixes
backjump-conds :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool
begin
inductive *backjump* **where**
trail *S* = *F' @ Marked K () # F*
 $\implies T \sim \text{prepend-trail } (\text{Propagated } L \text{ } ()) \text{ } (\text{reduce-trail-to}_{NOT} \text{ } F \text{ } S)$
 $\implies C \in \# \text{ clauses } S$
 $\implies \text{trail } S \models_{as} C \text{Not } C$
 $\implies \text{undefined-lit } F \text{ } L$
 $\implies \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' } (\text{lits-of } (\text{trail } S))$
 $\implies \text{clauses } S \models_{pm} C' + \{\#L\}$

```

 $\Rightarrow F \models_{as} CNot\ C'$ 
 $\Rightarrow backjump\text{-}conds\ C\ C'\ L\ S\ T$ 
 $\Rightarrow backjump\ S\ T$ 
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-clsesNOT remove-clsesNOT +
  propagate-ops trail clauses prepend-trail tl-trail add-clsesNOT remove-clsesNOT propagate-conds +
  decide-ops trail clauses prepend-trail tl-trail add-clsesNOT remove-clsesNOT +
  backjumping-ops trail clauses prepend-trail tl-trail add-clsesNOT remove-clsesNOT 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-clsesNOT remove-clsesNOT:: '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 clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool +
assumes
  bj-can-jump:
   $\bigwedge S\ C\ F'\ K\ F\ L.$ 
  inv S  $\Rightarrow$ 
  no-dup (trail S)  $\Rightarrow$ 
  trail S = F' @ Marked K () # F  $\Rightarrow$ 
  C  $\in$  # clauses S  $\Rightarrow$ 
  trail S  $\models_{as} CNot\ C$   $\Rightarrow$ 
  undefined-lit F L  $\Rightarrow$ 
  atm-of L  $\in$  atms-of-msu (clauses S)  $\cup$  atm-of ' (lits-of (F' @ Marked K () # F))  $\Rightarrow$ 
  clauses S  $\models_{pm} C' + \{\#L\#$   $\} \Rightarrow$ 
  F  $\models_{as} CNot\ C'$   $\Rightarrow$ 
   $\neg no\text{-}step\ backjump\ S$ 
begin

```

We cannot add a like condition $atms\text{-}of\ C' \subseteq atms\text{-}of\text{-}ms\ N$ because to ensure that we can backjump even if the last decision variable has disappeared.

The part of the condition $atm\text{-}of\ L \in atm\text{-}of\ ' lits\text{-}of\ (F' @ Marked\ K\ () \# F)$ is important, otherwise you are not sure that you can backtrack.

14.3.1 Definition

We define dpll with backjumping:

```

inductive dpll-bj :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
  bj-decideNOT: decideNOT S S'  $\Rightarrow$  dpll-bj S S' |
  bj-propagateNOT: propagateNOT S S'  $\Rightarrow$  dpll-bj S S' |
  bj-backjump: backjump S S'  $\Rightarrow$  dpll-bj S S'

```

lemmas *dpll-bj-induct* = *dpll-bj.induct*[*split-format*(*complete*)]

thm *dpll-bj-induct*[*OF dpll-with-backjumping-ops-axioms*]

lemma *dpll-bj-all-induct*[*consumes 2, case-names decide_{NOT} propagate_{NOT} backjump*]:
fixes *S T* :: '*st*

assumes

dpll-bj S T **and**

inv S

$\wedge L T. \text{undefined-lit } (\text{trail } S) L \implies \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S)$

$\implies T \sim \text{prepend-trail } (\text{Marked } L ()) S$

$\implies P S T$ **and**

$\wedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \implies \text{trail } S \models_{\text{as}} C \text{Not } C \implies \text{undefined-lit } (\text{trail } S) L$

$\implies T \sim \text{prepend-trail } (\text{Propagated } L ()) S$

$\implies P S T$ **and**

$\wedge C F' K F L C' T. C \in \# \text{ clauses } S \implies F' @ \text{Marked } K () \# F \models_{\text{as}} C \text{Not } C$

$\implies \text{trail } S = F' @ \text{Marked } K () \# F$

$\implies \text{undefined-lit } F L$

$\implies \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (F' @ \text{Marked } K () \# F))$

$\implies \text{clauses } S \models_{\text{pm}} C' + \{\#L\# \}$

$\implies F \models_{\text{as}} C \text{Not } C'$

$\implies T \sim \text{prepend-trail } (\text{Propagated } L ()) (\text{reduce-trail-to}_{\text{NOT}} F S)$

$\implies P S T$

shows *P S T*

apply (*induct T rule: dpll-bj-induct[OF local.dpll-with-backjumping-ops-axioms]*)

apply (*rule assms(1)*)

using *assms(3)* **apply** *blast*

apply (*elim propagateE*) **using** *assms(4)* **apply** *blast*

apply (*elim backjumpE*) **using** *assms(5)* *(inv S)* **by** *simp*

14.3.2 Basic properties

First, some better suited induction principle **lemma** *dpll-bj-clauses:*

assumes *dpll-bj S T* **and** *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_{NOT}-skip-beginning*)

Valuations **lemma** *dpll-bj-sat-iff:*

assumes *dpll-bj S T* **and** *inv S*

shows $I \models_{\text{sm}} \text{clauses } S \longleftrightarrow I \models_{\text{sm}} \text{clauses } T$

using *assms* **by** (*induction rule: dpll-bj-all-induct*) *auto*

Clauses **lemma** *dpll-bj-atms-of-ms-clauses-inv:*

assumes

dpll-bj S T **and**

inv S

shows $\text{atms-of-msu } (\text{clauses } S) = \text{atms-of-msu } (\text{clauses } T)$

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**

$\text{atm-of } ' (\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu } (\text{clauses } S)$

shows $\text{atm-of } \langle \text{lits-of } (\text{trail } T) \rangle \subseteq \text{atms-of-msu } (\text{clauses } S)$
using *assms* **by** (*induction rule: dpll-bj-all-induct*)
(auto simp: in-plus-implies-atm-of-on-atms-of-ms reduce-trail-to_{NOT}-skip-beginning)

lemma *dpll-bj-atms-in-trail-in-set:*

assumes *dpll-bj S T* **and**
inv S **and**
 $\text{atms-of-msu } (\text{clauses } S) \subseteq A$ **and**
 $\text{atm-of } \langle \text{lits-of } (\text{trail } S) \rangle \subseteq A$
shows $\text{atm-of } \langle \text{lits-of } (\text{trail } T) \rangle \subseteq A$
using *assms* **by** (*induction rule: dpll-bj-all-induct*)
(auto simp: in-plus-implies-atm-of-on-atms-of-ms)

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_{NOT}*

then show *?case* **using** *decomp* **by** *auto*

next

case (*propagate_{NOT} 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_{NOT}.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_{NOT}.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)  $\cup$  ( $\lambda m. \{\#lit\text{-}of\ m\# \}$ ) ‘set a  $\models_{ps}$ 
    ( $\lambda m. \{\#lit\text{-}of\ m\# \}$ ) ‘set a  $\cup$  ( $\lambda m. \{\#lit\text{-}of\ m\# \}$ ) ‘set (tl y)
    using a2 true-clss-clss-def by blast
  then show ( $\lambda m. \{\#lit\text{-}of\ m\# \}$ ) ‘set a  $\cup$  set-mset (clauses S)  $\models_{ps}$  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 ( $\lambda a. \{\#lit\text{-}of\ a\# \}$ ) ‘set a  $\cup$  set-mset ?N  $\models_{ps}$  ( $\lambda a. \{\#lit\text{-}of\ a\# \}$ ) ‘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 ( $\lambda a. \{\#lit\text{-}of\ a\# \}$ ) ‘set (fst (hd (get-all-marked-decomposition F)))
   $\cup$  set-mset (clauses S)
   $\models_{ps}$  ( $\lambda a. \{\#lit\text{-}of\ a\# \}$ ) ‘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 ‘F  $\models_{as}$  CNot D’ 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 ‘F  $\models_{as}$  CNot D’ 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 ‘F = b @ a’ 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 F = b @ a 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*:

length (get-all-marked-decomposition M) ≤ 1 + length M

by (*induction M rule: marked-lit-list-induct*) *auto*

lemma *length-in-get-all-marked-decomposition-bounded*:

assumes *i: i ∈ set (trail-weight S)*

shows *i ≤ Suc (length (trail S))*

proof –

obtain *a b* **where**

(a, b) ∈ set (get-all-marked-decomposition (trail S)) **and**

ib: i = Suc (length b)

using *i* **by** *auto*

then obtain *c* **where** *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:

- $1 + \text{card}(\text{atms-of-ms } A)$: $\text{card}(\text{atms-of-ms } A)$ 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.

- $2 + \text{card} (\text{atms-of-ms } A)$: $\text{card} (\text{atms-of-ms } A)$ is an upper bound on the number of elements, where adding one is necessary for the same reason as for the bound on the list, and one is needed to have a strict bound.

abbreviation *unassigned-lit* :: 'b literal multiset set \Rightarrow 'a list \Rightarrow nat **where**
unassigned-lit N M \equiv $\text{card} (\text{atms-of-ms } N) - \text{length } M$

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: $\text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

MA: $\text{atm-of } \text{'lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

n-d: $\text{no-dup} (\text{trail } S)$ **and**

finite: $\text{finite } A$

shows $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$

$> \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$

using *assms*(1,2)

proof (*induction rule*: *dpll-bj-all-induct*)

case (*propagate*_{NOT} C L) **note** CLN = *this*(1) **and** MC = *this*(2) **and** undef-L = *this*(3) **and** T = *this*(4)

have *incl*: $\text{atm-of } \text{'lits-of } (\text{Propagated } L ()) \# \text{trail } S \subseteq \text{atms-of-ms } A$

using *propagate*_{NOT}.*hyps* *propagate*-ops.*propagate*_{NOT} *dpll-bj-atms-in-trail-in-set* *bj-propagate*_{NOT}

NA MA CLN **by** (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

have *no-dup*: $\text{no-dup} (\text{Propagated } L ()) \# \text{trail } S$

using *defined-lit-map* n-d undef-L **by** *auto*

obtain a b l **where** M: $\text{get-all-marked-decomposition} (\text{trail } S) = (a, b) \# l$

by (*case-tac get-all-marked-decomposition* (trail S)) *auto*

have *b-le-M*: $\text{length } b \leq \text{length} (\text{trail } S)$

using *get-all-marked-decomposition-decomp*[of trail S] **by** (*simp add: M*)

have *finite* ($\text{atms-of-ms } A$) **using** *finite* **by** *simp*

then have $\text{length} (\text{Propagated } L ()) \# \text{trail } S \leq \text{card} (\text{atms-of-ms } A)$

using *incl* *finite* **unfolding** *no-dup-length-eq-card-atm-of-lits-of*[OF *no-dup*]

by (*simp add: card-mono*)

then have *latm*: $\text{unassigned-lit } A \ b = \text{Suc} (\text{unassigned-lit } A (\text{Propagated } L \ d \ \# \ b))$

using *b-le-M* **by** *auto*

then show ?*case* **using** T undef-L **by** (*auto simp: latm M* μ_C -*cons*)

next

case (*decide*_{NOT} L) **note** undef-L = *this*(1) **and** MC = *this*(2) **and** T = *this*(3)

have *incl*: $\text{atm-of } \text{'lits-of } (\text{Marked } L ()) \# (\text{trail } S) \subseteq \text{atms-of-ms } A$

using *dpll-bj-atms-in-trail-in-set* *bj-decide*_{NOT} *decide*_{NOT}.*decide*_{NOT}[OF *decide*_{NOT}.*hyps*] NA MA

MC

by *auto*

have *no-dup*: $\text{no-dup} (\text{Marked } L ()) \# (\text{trail } S)$

using *defined-lit-map* n-d undef-L **by** *auto*

obtain a b l **where** M: $\text{get-all-marked-decomposition} (\text{trail } S) = (a, b) \# l$

by (*case-tac get-all-marked-decomposition* (trail S)) *auto*

then have $\text{length} (\text{Marked } L ()) \# (\text{trail } S) \leq \text{card} (\text{atms-of-ms } A)$

```

    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)  $\subseteq$  atms-of-ms A
    using dpll-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  $\leq$  length (trail S)
    using get-all-marked-decomposition-decomp[of trail S] by (simp add: M)
  have fin-atms-A: finite (atms-of-ms A) using finite by simp

  then have F-le-A: length (Propagated L () # F)  $\leq$  card (atms-of-ms 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)  $\leq$  (card (atms-of-ms 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 ( $\lambda a. \text{Suc} (\text{length} (\text{snd } a))$ ) (rev (get-all-marked-decomposition (F' @ Marked K () # F)))
    = map ( $\lambda a. \text{Suc} (\text{length} (\text{snd } a))$ ) (rev (get-all-marked-decomposition F)) @ rem
    using take-length-get-all-marked-decomposition-marked-sandwich[of F  $\lambda a. \text{Suc} (\text{length } a)$  F' K]
    unfolding o-def by (metis append-take-drop-id)
  then have rem: map ( $\lambda a. \text{Suc} (\text{length} (\text{snd } a))$ )
    (get-all-marked-decomposition (F' @ Marked K () # F))
    = rev rem @ map ( $\lambda a. \text{Suc} (\text{length} (\text{snd } a))$ ) ((get-all-marked-decomposition F))
    by (simp add: rev-map[symmetric] rev-swap)
  have length (rev rem @ map ( $\lambda a. \text{Suc} (\text{length} (\text{snd } a))$ ) (get-all-marked-decomposition F))
     $\leq$  Suc (card (atms-of-ms 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  $\implies$  length xs - Suc i < length xs
      by auto
    then have H: i < length xs  $\implies$  rev xs ! i  $\in$  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  $\forall i < \text{length } \text{rem}. \text{rev rem} ! i < \text{card} (\text{atms-of-ms } A) + 2$ 
    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 μ_C -bounded[of rev rem card (atms-of-ms A)+2 unassigned-lit A l] T undef-L
 by (simp add: rem μ_C -append μ_C -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-msu (clauses S) \subseteq atms-of-ms A and
 M-A: atm-of ' lits-of (trail S) \subseteq atms-of-ms A and
 nd: no-dup (trail S) and
 fin-A: finite A
 shows $(2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)})$
 $- \mu_C (1 + \text{card (atms-of-ms A)}) (2 + \text{card (atms-of-ms A)}) (\text{trail-weight T})$
 $< (2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)})$
 $- \mu_C (1 + \text{card (atms-of-ms A)}) (2 + \text{card (atms-of-ms A)}) (\text{trail-weight S})$

proof -

let ?b = 2 + card (atms-of-ms A)
 let ?s = 1 + card (atms-of-ms A)
 let ? μ = μ_C ?s ?b
 have M'-A: atm-of ' lits-of (trail T) \subseteq atms-of-ms 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 \implies length xs - Suc i < length xs
 by auto
 then have H: i < length xs \implies xs ! i \in 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) \leq card (atms-of-ms 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) \leq card (atms-of-ms 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) \leq 1 + card (atms-of-ms A)
 using l-M'-A length-get-all-marked-decomposition-length[of trail T] by auto
 have bounded-M: $\forall i < \text{length (trail-weight T)}. (\text{trail-weight T})! i < \text{card (atms-of-ms A)} + 2$
 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 dpll-bj-trail-mes-increasing-prop[OF dpll inv N-A M-A nd fin-A]

have μ_C ?s ?b (trail-weight S) < μ_C ?s ?b (trail-weight T) by simp

moreover from μ_C -bounded[OF bounded-M l-trail-weight-M]

have μ_C ?s ?b (trail-weight T) \leq ?b \wedge ?s by auto

ultimately show ?thesis by linarith

qed

lemma wf-dpll-bj:

assumes fin: finite A

shows wf {(T, S). dpll-bj S T

\wedge atms-of-msu (clauses S) \subseteq atms-of-ms A \wedge atm-of ' lits-of (trail S) \subseteq atms-of-ms A

\wedge no-dup (trail S) \wedge inv S}

(is wf ?A)

proof (rule wf-bounded-measure[of -

```

    λ-. (2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
    λS. μC (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight S)]
fix a b :: 'st
let ?b = 2 + card (atms-of-ms A)
let ?s = 1 + card (atms-of-ms A)
let ?μ = μC ?s ?b
assume ab: (b, a) ∈ {(T, S). dpll-bj S T
  ∧ atms-of-msu (clauses S) ⊆ atms-of-ms A ∧ atm-of ' lits-of (trail S) ⊆ atms-of-ms A
  ∧ no-dup (trail S) ∧ inv S}

have fin-A: finite (atms-of-ms A)
  using fin by auto
have
  dpll-bj: dpll-bj a b and
  N-A: atms-of-msu (clauses a) ⊆ atms-of-ms A and
  M-A: atm-of ' lits-of (trail a) ⊆ atms-of-ms 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-ms A
  by (meson M-A N-A ⟨dpll-bj a b⟩ dpll-bj-atms-in-trail-in-set inv)
have nd': no-dup (trail b)
  using ⟨dpll-bj a b⟩ 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 a) ≤ card (atms-of-ms 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-ms 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-ms 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-ms 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 $A :: 'v$ literal multiset set **and** $S\ T :: 'st$

assumes

atms-of-msu (*clauses* S) \subseteq *atms-of-ms* A **and**

atm-of ' *lits-of* (*trail* S) \subseteq *atms-of-ms* A **and**

no-dup (*trail* S) **and**

finite A **and**

inv: *inv* S **and**

n-s: *no-step dpll-bj* S **and**

decomp: *all-decomposition-implies-m* (*clauses* 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 $?N = \text{set-mset} (\text{clauses } S)$

let $?M = \text{trail } S$

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** $M = \text{this}(2)$

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-ms* $?N$

using *sat* **unfolding** *satisfiable-def-min* **by** *auto*

let $?I = I \cup \{P \mid P. P \in \text{lits-of } ?M \wedge \text{atm-of } P \notin \text{atm-of ' } I\}$

let $?O = \{\{\# \text{lit-of } L \# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M \wedge \text{atm-of } (\text{lit-of } L) \notin \text{atms-of-ms } ?N\}$

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-ms ?N ⊆ atm-of ' lits-of ?M
  proof (rule ccontr)
    assume ¬ ?thesis
    then obtain l :: 'v where
      l-N: l ∈ atms-of-ms ?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))
    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 ⊨as CNot C
  by (metis ⟨C ∈ set-mset (clauses S)⟩ ⟨¬ trail S ⊨a C⟩ all-variables-defined-not-imply-cnot
    atms-N-M atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of atms-of-ms-CNot-atms-of-ms
    subset-eq)
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-ms ?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-ms-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-ms ?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 ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ cons-I' atms-N-M
      by (meson ⟨trail S ⊨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 literal and
  F F' :: ('v, unit, unit) marked-lit list where
  M-K: ?M = F' @ 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 = Marked K () :: ('v, unit, unit) marked-lit
have ?K  $\in$  set ?M
unfolding M-K by auto
let ?C = image-mset lit-of {#L  $\in$  #mset ?M. is-marked L  $\wedge$  L  $\neq$  ?K#} :: 'v literal multiset
let ?C' = set-mset (image-mset ( $\lambda L :: 'v$  literal. {#L#}) (?C + {#lit-of ?K#}))
have ?N  $\cup$  {#{#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set ?M}  $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set ?M
using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' = {#{#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  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#}) ' set ?M
by auto
have N-M-False: ?N  $\cup$  ( $\lambda L.$  {#lit-of L#}) ' (set ?M)  $\models_{ps}$  {#{#}}
using M  $\langle$  ?M  $\models_{as}$  CNot C  $\rangle$  ( $C \in ?N$ ) 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-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#}) ' set ?M =
?N  $\cup$  ( $\lambda a.$  {#lit-of a#}) ' 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  $\models_p$  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-ms (?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  $H = \text{this}$ 

  have  $\neg I \models_s ?C'$ 
    using  $\langle ?N \cup ?C' \models_{ps} \{\{\#\}\} \rangle \text{ tot cons } \langle I \models_s ?N \rangle$ 
    unfolding true-clss-clss-def total-over-m-def
    by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of)
  then show  $I \models \text{image-mset } \text{uminus } ?C + \{\# - K \#\}$ 
    unfolding true-clss-def true-cl-def Bex-mset-def
    using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
    by (auto dest!: H)
  qed
moreover have  $F \models_{as} CNot (\text{image-mset } \text{uminus } ?C)$ 
  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  $\{\# L : \# \text{ mset } ?M. \text{is-marked } L \wedge L \neq \text{Marked } K ()\#\}$ )
     $\langle C \in ?N \rangle n-s \langle ?M \models_{as} CNot C \rangle \text{ bj-backjump inv } \langle \text{no-dup } (\text{trail } S) \rangle$ 
    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-clNOT remove-clNOT
  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-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 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-ms-clauses-inv:
  assumes
    dpll-bj** S T and inv S

```

shows $\text{atms-of-msu}(\text{clauses } S) = \text{atms-of-msu}(\text{clauses } T)$
using *assms* **by** (*induction rule*: *rtranclp-induct*)
 (*auto dest*: *rtranclp-dpll-bj-inv dpll-bj-atms-of-ms-clauses-inv*)

lemma *rtranclp-dpll-bj-atms-in-trail*:

assumes
 $\text{dpll-bj}^{**} S T$ **and**
 $\text{inv } S$ **and**
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu}(\text{clauses } S)$
shows $\text{atm-of } '(\text{lits-of } (\text{trail } T)) \subseteq \text{atms-of-msu}(\text{clauses } T)$
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
using *dpll-bj-atms-in-trail dpll-bj-atms-of-ms-clauses-inv rtranclp-dpll-bj-inv* **by** *auto*

lemma *rtranclp-dpll-bj-sat-iff*:

assumes $\text{dpll-bj}^{**} S T$ **and** $\text{inv } S$
shows $I \models_{\text{sm}} \text{clauses } S \longleftrightarrow I \models_{\text{sm}} \text{clauses } T$
using *assms* **by** (*induction rule*: *rtranclp-induct*)
 (*auto dest*!: *dpll-bj-sat-iff simp: rtranclp-dpll-bj-inv*)

lemma *rtranclp-dpll-bj-atms-in-trail-in-set*:

assumes
 $\text{dpll-bj}^{**} S T$ **and**
 $\text{inv } S$
 $\text{atms-of-msu}(\text{clauses } S) \subseteq A$ **and**
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq A$
shows $\text{atm-of } '(\text{lits-of } (\text{trail } T)) \subseteq A$
using *assms*
by (*induction rule*: *rtranclp-induct*)
 (*auto dest*: *rtranclp-dpll-bj-inv*
simp add: dpll-bj-atms-in-trail-in-set rtranclp-dpll-bj-atms-of-ms-clauses-inv
rtranclp-dpll-bj-inv)

lemma *rtranclp-dpll-bj-all-decomposition-implies-inv*:

assumes
 $\text{dpll-bj}^{**} S T$ **and**
 $\text{inv } S$
 $\text{all-decomposition-implies-m}(\text{clauses } S) (\text{get-all-marked-decomposition } (\text{trail } S))$
shows $\text{all-decomposition-implies-m}(\text{clauses } T) (\text{get-all-marked-decomposition } (\text{trail } T))$
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*:

$\{(T, S). \text{dpll-bj}^{++} S T$
 $\wedge \text{atms-of-msu}(\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$
 $\subseteq \{(T, S). \text{dpll-bj } S T \wedge \text{atms-of-msu}(\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A \wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}^+$
 (*is* $?A \subseteq ?B^+$)

proof *standard*

fix x
assume $x-A: x \in ?A$
obtain $S T::'st$ **where**
 $x[\text{simp}]: x = (T, S)$ **by** (*cases* x) *auto*
have
 $\text{dpll-bj}^{++} S T$ **and**

$atms\text{-}of\text{-}msu\ (clauses\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $atm\text{-}of\ ' lits\text{-}of\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $no\text{-}dup\ (trail\ S)$ **and**
 $inv\ S$
using $x\text{-}A$ **by** *auto*
then show $x \in ?B^+$ **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 $N\text{-}A = this(4)$ **and** $M\text{-}A = this(5)$ **and** $nd = this(6)$ **and** $inv = this(7)$

have [*simp*]: $atms\text{-}of\text{-}msu\ (clauses\ S) = atms\text{-}of\text{-}msu\ (clauses\ T)$
using $step\ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv\ tranclp\text{-}into\text{-}rtranclp\ inv$ **by** *fastforce*
have $no\text{-}dup\ (trail\ T)$
using $local.step\ nd\ rtranclp\text{-}dpll\text{-}bj\text{-}no\text{-}dup\ tranclp\text{-}into\text{-}rtranclp\ inv$ **by** *fastforce*
moreover have $atm\text{-}of\ ' (lits\text{-}of\ (trail\ T)) \subseteq atms\text{-}of\text{-}ms\ A$
by ($metis\ inv\ M\text{-}A\ N\text{-}A\ local.step\ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set\ tranclp\text{-}into\text{-}rtranclp$)
moreover have $inv\ T$
using $inv\ local.step\ rtranclp\text{-}dpll\text{-}bj\text{-}inv\ tranclp\text{-}into\text{-}rtranclp$ **by** *fastforce*
ultimately have $(U, T) \in ?B$ **using** $ST\ N\text{-}A\ M\text{-}A\ inv$ **by** *auto*
then show $?case$ **using** IH **by** (*rule tranclp-into-tranclp2*)
qed
qed

lemma *wf-tranclp-dpll-bj*:
assumes $fin: finite\ A$
shows $wf\ \{(T, S). dpll\text{-}bj^{++}\ S\ T$
 $\wedge atms\text{-}of\text{-}msu\ (clauses\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ ' lits\text{-}of\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
 $\wedge no\text{-}dup\ (trail\ S) \wedge inv\ S\}$
using $wf\text{-}tranclp[OF\ wf\text{-}dpll\text{-}bj[OF\ fin]]\ rtranclp\text{-}dpll\text{-}bj\text{-}inv\text{-}incl\text{-}dpll\text{-}bj\text{-}inv\text{-}tranclp$
by (*rule wf-subset*)

lemma *dpll-bj-sat-ext-iff*:
 $dpll\text{-}bj\ S\ T \implies inv\ S \implies I \models_{sextm} clauses\ S \longleftrightarrow I \models_{sextm} clauses\ T$
by (*simp add: dpll-bj-clauses*)

lemma *rtranclp-dpll-bj-sat-ext-iff*:
 $dpll\text{-}bj^{**}\ S\ T \implies inv\ S \implies I \models_{sextm} clauses\ S \longleftrightarrow I \models_{sextm} clauses\ T$
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
 $full: full\ dpll\text{-}bj\ S\ T$ **and**
 $atms\text{-}S: atms\text{-}of\text{-}msu\ (clauses\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $atms\text{-}trail: atm\text{-}of\ ' lits\text{-}of\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ **and**
 $finite\ A$ **and**
 $inv: inv\ S$ **and**
 $decomp: all\text{-}decomposition\text{-}implies\text{-}m\ (clauses\ S)\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S))$
shows $unsatisfiable\ (set\text{-}mset\ (clauses\ S))$
 $\vee (trail\ T \models_{asm} clauses\ S \wedge satisfiable\ (set\text{-}mset\ (clauses\ S)))$

proof –

have $st: dpll\text{-}bj^{**} S T$ **and** $no\text{-}step\ dpll\text{-}bj\ T$
using $full\ unfolding\ full\text{-}def$ **by** $fast+$
moreover **have** $atms\text{-}of\text{-}msu\ (clauses\ T) \subseteq atms\text{-}of\text{-}ms\ A$
using $atms\text{-}S\ inv\ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv\ st$ **by** $blast$
moreover **have** $atm\text{-}of\ 'lits\text{-}of\ (trail\ T) \subseteq atms\text{-}of\text{-}ms\ A$
using $atms\text{-}S\ atms\text{-}trail\ inv\ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set\ st$ **by** $auto$
moreover **have** $no\text{-}dup\ (trail\ T)$
using $n\text{-}d\ inv\ rtranclp\text{-}dpll\text{-}bj\text{-}no\text{-}dup\ st$ **by** $blast$
moreover **have** $inv: inv\ T$
using $inv\ rtranclp\text{-}dpll\text{-}bj\text{-}inv\ st$ **by** $blast$
moreover
have $decomp: all\text{-}decomposition\text{-}implies\text{-}m\ (clauses\ T)\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ T))$
using $\langle inv\ S \rangle\ decomp\ rtranclp\text{-}dpll\text{-}bj\text{-}all\text{-}decomposition\text{-}implies\text{-}inv\ st$ **by** $blast$
ultimately **have** $unsatisfiable\ (set\text{-}mset\ (clauses\ T))$
 $\vee\ (trail\ T \models_{asm}\ clauses\ T \wedge\ satisfiable\ (set\text{-}mset\ (clauses\ T)))$
using $\langle finite\ A \rangle\ dpll\text{-}backjump\text{-}final\text{-}state$ **by** $force$
then **show** $?thesis$
by $(meson\ \langle inv\ S \rangle\ rtranclp\text{-}dpll\text{-}bj\text{-}sat\text{-}iff\ satisfiable\text{-}carac\ st\ true\text{-}annots\text{-}true\text{-}cls)$
qed

corollary $full\text{-}dpll\text{-}backjump\text{-}final\text{-}state\text{-}from\text{-}init\text{-}state$:

fixes $A :: 'v\ literal\ multiset\ set$ **and** $S\ T :: 'st$
assumes
 $full: full\ dpll\text{-}bj\ S\ T$ **and**
 $trail\ S = []$ **and**
 $clauses\ S = N$ **and**
 $inv\ S$
shows $unsatisfiable\ (set\text{-}mset\ N) \vee\ (trail\ T \models_{asm}\ N \wedge\ satisfiable\ (set\text{-}mset\ N))$
using $assms\ full\text{-}dpll\text{-}backjump\text{-}final\text{-}state[of\ S\ T\ set\text{-}mset\ N]$ **by** $auto$

lemma $tranclp\text{-}dpll\text{-}bj\text{-}trail\text{-}mes\text{-}decreasing\text{-}prop$:

assumes $dpll: dpll\text{-}bj^{++} S\ T$ **and** $inv: inv\ S$ **and**
 $N\text{-}A: atms\text{-}of\text{-}msu\ (clauses\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $M\text{-}A: atm\text{-}of\ 'lits\text{-}of\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ **and**
 $fin\text{-}A: finite\ A$
shows $(2 + card\ (atms\text{-}of\text{-}ms\ A)) \wedge\ (1 + card\ (atms\text{-}of\text{-}ms\ A))$
 $\quad -\ \mu_C\ (1 + card\ (atms\text{-}of\text{-}ms\ A))\ (2 + card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)$
 $\quad <\ (2 + card\ (atms\text{-}of\text{-}ms\ A)) \wedge\ (1 + card\ (atms\text{-}of\text{-}ms\ A))$
 $\quad -\ \mu_C\ (1 + card\ (atms\text{-}of\text{-}ms\ A))\ (2 + card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ S)$
using $dpll$

proof $(induction)$

case $base$

then **show** $?case$

using $N\text{-}A\ M\text{-}A\ n\text{-}d\ dpll\text{-}bj\text{-}trail\text{-}mes\text{-}decreasing\text{-}prop\ fin\text{-}A\ inv$ **by** $blast$

next

case $(step\ T\ U)$ **note** $st = this(1)$ **and** $dpll = this(2)$ **and** $IH = this(3)$

have $atms\text{-}of\text{-}msu\ (clauses\ S) = atms\text{-}of\text{-}msu\ (clauses\ T)$

using $rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv$ **by** $(metis\ dpll\text{-}bj\text{-}clauses\ dpll\text{-}bj\text{-}inv\ inv\ st\ tranclpD)$

then **have** $N\text{-}A': atms\text{-}of\text{-}msu\ (clauses\ T) \subseteq atms\text{-}of\text{-}ms\ A$

using $N\text{-}A$ **by** $auto$

moreover **have** $M\text{-}A': atm\text{-}of\ 'lits\text{-}of\ (trail\ T) \subseteq atms\text{-}of\text{-}ms\ A$

by $(meson\ M\text{-}A\ N\text{-}A\ inv\ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set\ st\ dpll)$

```

    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  $\implies$  atms-of C  $\subseteq$  atms-of-msu (clauses S)  $\cup$  atm-of ' (lits-of (trail S))
   $\implies$  learn-cond C S
   $\implies$  T  $\sim$  add-clsNOT C S
   $\implies$  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-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
  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
   $\implies$  forget-cond C S
   $\implies$  C  $\in \#$  clauses S
   $\implies$  T  $\sim$  remove-clsNOT C S
   $\implies$  forgetNOT S T
inductive-cases forgetE: forgetNOT S T

```

```

lemma forget- $\mu_C$ -stable:
  assumes forgetNOT S T
  shows  $\mu_C A B (\text{trail-weight } S) = \mu_C A B (\text{trail-weight } T)$ 
  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  $\Longrightarrow$  learn-and-forgetNOT S T |
  lf-forget: forgetNOT S T  $\Longrightarrow$  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 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'  $\Longrightarrow$  cdclNOT S S' |
  c-learn: learn S S'  $\Longrightarrow$  cdclNOT S S' |
  c-forgetNOT: forgetNOT S S'  $\Longrightarrow$  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:  $\bigwedge T. \text{dpll-bj } S T \Longrightarrow P S T$  and
  learning:
     $\bigwedge C T. \text{clauses } S \models_{pm} C \Longrightarrow$ 
     $\text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S)) \Longrightarrow$ 
     $T \sim \text{add-cl}_{NOT} C S \Longrightarrow$ 

```


$P S T$ and
forgetting: $\bigwedge C T$. clauses S – replicate-mset (count (clauses S) C) $C \models_{pm} C \implies$
 $C \in \# \text{ clauses } S \implies$
 $T \sim \text{remove-cl}_S \text{ NOT } C S \implies$
 $P S T$
shows $P S T$
using *assms(1) by (induction rule: $cdcl_{NOT}.induct$)*
(auto intro: assms(2, 3, 4) elim!: learnE forgetE)+

lemma $cdcl_{NOT}$ -no-dup:

assumes
 $cdcl_{NOT} S T$ and
 $inv S$ and
 $no\text{-}dup \text{ (trail } S)$
shows *$no\text{-}dup \text{ (trail } T)$*
using *assms by (induction rule: $cdcl_{NOT}$ -all-induct) (auto intro: $dpll\text{-}bj\text{-}no\text{-}dup$)*

Consistency of the trail lemma $cdcl_{NOT}$ -consistent:

assumes
 $cdcl_{NOT} S T$ and
 $inv S$ and
 $no\text{-}dup \text{ (trail } S)$
shows *$consistent\text{-}interp \text{ (lits-of (trail } T))$*
using *$cdcl_{NOT}$ -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 $cdcl_{NOT}$ -atms-of-ms-clauses-decreasing:

assumes *$cdcl_{NOT} S T$ and $inv S$ and $no\text{-}dup \text{ (trail } S)$*
shows *$atms\text{-}of\text{-}msu \text{ (clauses } T) \subseteq atms\text{-}of\text{-}msu \text{ (clauses } S) \cup atm\text{-}of \text{ ' (lits-of (trail } S))$*
using *assms by (induction rule: $cdcl_{NOT}$ -all-induct)*
(auto dest!: $dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv \text{ set-mp simp add: } atms\text{-}of\text{-}ms\text{-}def \text{ Union-eq}$)

lemma $cdcl_{NOT}$ -atms-in-trail:

assumes *$cdcl_{NOT} S T$ and $inv S$ and $no\text{-}dup \text{ (trail } S)$*
and *$atm\text{-}of \text{ ' (lits-of (trail } S)) \subseteq atms\text{-}of\text{-}msu \text{ (clauses } S)$*
shows *$atm\text{-}of \text{ ' (lits-of (trail } T)) \subseteq atms\text{-}of\text{-}msu \text{ (clauses } S)$*
using *assms by (induction rule: $cdcl_{NOT}$ -all-induct) (auto simp add: $dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail$)*

lemma $cdcl_{NOT}$ -atms-in-trail-in-set:

assumes
 $cdcl_{NOT} S T$ and $inv S$ and $no\text{-}dup \text{ (trail } S)$ and
 $atms\text{-}of\text{-}msu \text{ (clauses } S) \subseteq A$ and
 $atm\text{-}of \text{ ' (lits-of (trail } S)) \subseteq A$
shows *$atm\text{-}of \text{ ' (lits-of (trail } T)) \subseteq A$*
using *assms*
by *(induction rule: $cdcl_{NOT}$ -all-induct)*
(simp-all add: $dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set \text{ } dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv$)

lemma $cdcl_{NOT}$ -all-decomposition-implies:

assumes *$cdcl_{NOT} S T$ and $inv S$ and $n\text{-}d[simp]$: $no\text{-}dup \text{ (trail } S)$ and*
 $all\text{-}decomposition\text{-}implies\text{-}m \text{ (clauses } S) \text{ (get-all-marked-decomposition (trail } S))$
shows
 $all\text{-}decomposition\text{-}implies\text{-}m \text{ (clauses } T) \text{ (get-all-marked-decomposition (trail } T))$

```

using assms(1,2,4)
proof (induction rule: cdclNOT-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 (forgetNOT C T) note cls-C = this(1) and C = this(2) and T = this(3) and inv = 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 ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set a  $\cup$  set-mset (clauses S)  $\models_{ps}$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ 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-clsNOT cls-C insert-Diff order-refl
      set-mset-minus-replicate-mset(1) state-eqNOT-clauses true-clss-clss-def
      true-clss-clss-insert)
    ultimately show ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set a  $\cup$  set-mset (clauses T)
       $\models_{ps}$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set b
      using true-clss-clss-generalise-true-clss-clss by blast
  qed
qed

```

Extension of models **lemma** *cdcl_{NOT}-bj-sat-ext-iff*:

assumes *cdcl_{NOT} S T* **and** *inv S* **and** *n-d: no-dup (trail S)*

shows $I \models_{sextm} \text{clauses } S \longleftrightarrow I \models_{sextm} \text{clauses } T$

using *assms*

proof (*induction rule: cdcl_{NOT}-all-induct*)

case *dpll-bj*

then show ?*case* **by** (*simp add: dpll-bj-clauses*)

next

case (*learn C T*) **note** *T = this(3)*

{ fix *J*

assume

I \models_{sextm} *clauses S* **and**

I \subseteq *J* **and**

*tot: total-over-m J (set-mset ($\{\#C\# \} + (\text{clauses } S)))$ **and***

cons: consistent-interp J

then have *J* \models_{sm} *clauses S* **unfolding** *true-clss-ext-def* **by** *auto*

moreover

with $\langle \text{clauses } S \models_{pm} C \rangle$ **have** *J* \models *C*

using *tot cons* **unfolding** *true-clss-clss-def* **by** *auto*

ultimately have *J* \models_{sm} $\{\#C\# \} + \text{clauses } S$ **by** *auto*

}

then have *H: I* $\models_{sextm} (\text{clauses } S) \implies I \models_{sext} \text{insert } C (\text{set-mset } (\text{clauses } S))$

```

  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 (forgetNOT C T) note cls-C = this(1) and T = this(3)
{ fix J
  assume
    I  $\models_{\text{set}}$  set-mset (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  $\models_s$  set-mset (clauses S) - {C}
    unfolding true-clss-ext-def by (meson Diff-subset total-over-m-subset)

  moreover
  with cls-C have J  $\models$  C
    using tot cons unfolding true-clss-cls-def
    by (metis Un-commute forgetNOT.hyps(2) insert-Diff insert-is-Un mem-set-mset-iff order-refl
      set-mset-minus-replicate-mset(1))
  ultimately have J  $\models_{sm}$  (clauses S) by (metis insert-Diff-single true-clss-insert)
}
then have H: I  $\models_{\text{set}}$  set-mset (clauses S) - {C}  $\implies$  I  $\models_{\text{setm}}$  (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 cdclNOT-inv:  $\bigwedge S T. \text{cdcl}_{\text{NOT}} S T \implies \text{inv } S \implies \text{inv } T$ 
begin
sublocale dpll-with-backjumping
  apply unfold-locales
  using cdclNOT.simps cdclNOT-inv by auto

lemma rtranclp-cdclNOT-inv:
  cdclNOT** S T  $\implies$  inv S  $\implies$  inv T
  by (induction rule: rtranclp-induct) (auto simp add: cdclNOT-inv)

lemma rtranclp-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: rtranclp-induct) (auto intro: cdclNOT-no-dup rtranclp-cdclNOT-inv)

lemma rtranclp-cdclNOT-trail-clauses-bound:
  assumes
    cdcl: cdclNOT** S T and
    inv: inv S and

```

$n\text{-d}$: $\text{no-dup } (\text{trail } S)$ **and**
 $\text{atms-clauses-}S$: $\text{atms-of-msu } (\text{clauses } S) \subseteq A$ **and**
 $\text{atms-trail-}S$: $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq A$
shows $\text{atm-of } '(\text{lits-of } (\text{trail } T)) \subseteq A \wedge \text{atms-of-msu } (\text{clauses } T) \subseteq A$
using cdcl
proof (*induction rule: rtrancpl-induct*)
case base
then show $?case$ **using** $\text{atms-clauses-}S$ $\text{atms-trail-}S$ **by** simp
next
case ($\text{step } T \ U$) **note** $st = \text{this}(1)$ **and** $\text{cdcl}_{NOT} = \text{this}(2)$ **and** $IH = \text{this}(3)$
have $\text{inv } T$ **using** $\text{inv } st$ $\text{rtrancpl-cdcl}_{NOT}\text{-inv}$ **by** blast
have $\text{no-dup } (\text{trail } T)$
using $\text{rtrancpl-cdcl}_{NOT}\text{-no-dup}[\text{of } S \ T]$ st cdcl_{NOT} $\text{inv } n\text{-d}$ **by** blast
then have $\text{atms-of-msu } (\text{clauses } U) \subseteq A$
using $\text{cdcl}_{NOT}\text{-atms-of-ms-clauses-decreasing}[\text{OF } \text{cdcl}_{NOT}]$ IH $n\text{-d}$ $\langle \text{inv } T \rangle$ **by** auto
moreover
have $\text{atm-of } '(\text{lits-of } (\text{trail } U)) \subseteq A$
using $\text{cdcl}_{NOT}\text{-atms-in-trail-in-set}[\text{OF } \text{cdcl}_{NOT}, \text{ of } A]$ $\langle \text{no-dup } (\text{trail } T) \rangle$
by ($\text{meson } \text{atms-trail-}S$ $\text{atms-clauses-}S$ IH $\langle \text{inv } T \rangle$ cdcl_{NOT})
ultimately show $?case$ **by** fast
qed

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-all-decomposition-implies}$:
assumes $\text{cdcl}_{NOT}^{**} \ S \ T$ **and** $\text{inv } S$ **and** $\text{no-dup } (\text{trail } S)$ **and**
 $\text{all-decomposition-implies-m } (\text{clauses } S)$ ($\text{get-all-marked-decomposition } (\text{trail } S)$)
shows
 $\text{all-decomposition-implies-m } (\text{clauses } T)$ ($\text{get-all-marked-decomposition } (\text{trail } T)$)
using assms **by** (*induction*)
(auto intro: rtrancpl-cdcl_{NOT}-inv cdcl_{NOT}-all-decomposition-implies rtrancpl-cdcl_{NOT}-no-dup)

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-bj-sat-ext-iff}$:
assumes $\text{cdcl}_{NOT}^{**} \ S \ T$ **and** $\text{inv } S$ **and** $\text{no-dup } (\text{trail } S)$
shows $I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$
using assms **apply** (*induction rule: rtrancpl-induct*)
using $\text{cdcl}_{NOT}\text{-bj-sat-ext-iff}$ **by** (*auto intro: rtrancpl-cdcl_{NOT}-inv rtrancpl-cdcl_{NOT}-no-dup*)

definition $\text{cdcl}_{NOT}\text{-NOT-all-inv}$ **where**
 $\text{cdcl}_{NOT}\text{-NOT-all-inv } A \ S \longleftrightarrow (\text{finite } A \wedge \text{inv } S \wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of } ' \text{lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{no-dup } (\text{trail } S))$

lemma $\text{cdcl}_{NOT}\text{-NOT-all-inv}$:
assumes $\text{cdcl}_{NOT}^{**} \ S \ T$ **and** $\text{cdcl}_{NOT}\text{-NOT-all-inv } A \ S$
shows $\text{cdcl}_{NOT}\text{-NOT-all-inv } A \ T$
using assms **unfolding** $\text{cdcl}_{NOT}\text{-NOT-all-inv-def}$
by (*simp add: rtrancpl-cdcl_{NOT}-inv rtrancpl-cdcl_{NOT}-no-dup rtrancpl-cdcl_{NOT}-trail-clauses-bound*)

abbreviation learn-or-forget **where**
 $\text{learn-or-forget } S \ T \equiv (\lambda S \ T. \text{learn } S \ T \vee \text{forget}_{NOT} \ S \ T) \ S \ T$

lemma $\text{rtrancpl-learn-or-forget-cdcl}_{NOT}$:
 $\text{learn-or-forget}^{**} \ S \ T \implies \text{cdcl}_{NOT}^{**} \ S \ T$
using $\text{rtrancpl-mono}[\text{of } \text{learn-or-forget } \text{cdcl}_{NOT}]$ $\text{cdcl}_{NOT}.c\text{-learn}$ $\text{cdcl}_{NOT}.c\text{-forget}_{NOT}$ **by** blast

lemma $\text{learn-or-forget-dpll-}\mu_C$:

assumes
l-f: *learn-or-forget*** *S T* **and**
dpll: *dpll-bj* *T U* **and**
inv: *cdcl*_{NOT}-*NOT-all-inv* *A S*
shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } U)$
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$
(is $?_{\mu} U < ?_{\mu} S$ **)**
proof –
have $?_{\mu} S = ?_{\mu} T$
using *l-f*
proof (*induction*)
case *base*
then show *?case* **by** *simp*
next
case (*step* *T U*)
moreover then have *no-dup* (*trail* *T*)
using *rtrancpl-cdcl*_{NOT}-*no-dup*[*of S T*] *cdcl*_{NOT}-*NOT-all-inv-def inv*
*rtrancpl-learn-or-forget-cdcl*_{NOT} **by** *auto*
ultimately show *?case*
using *forget- μ_C -stable learn- μ_C -stable inv* **unfolding** *cdcl*_{NOT}-*NOT-all-inv-def* **by** *presburger*
qed
moreover have *cdcl*_{NOT}-*NOT-all-inv* *A T*
using *rtrancpl-learn-or-forget-cdcl*_{NOT} *cdcl*_{NOT}-*NOT-all-inv* *l-f inv* **by** *blast*
ultimately show *?thesis*
using *dpll-bj-trail-mes-decreasing-prop*[*of T U A, OF dpll*] *finite*
unfolding *cdcl*_{NOT}-*NOT-all-inv-def* **by** *linarith*
qed

lemma *infinite-cdcl*_{NOT}-*exists-learn-and-forget-infinite-chain*:

assumes
 $\bigwedge i. \text{cdcl}_{NOT} (f i) (f (\text{Suc } i))$ **and**
inv: *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* $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (f 0))$
arbitrary: *f*
rule: *nat-less-induct-case*)
case (*Suc n*) **note** *IH* = *this*(1) **and** μ = *this*(2) **and** *cdcl*_{NOT} = *this*(3) **and** *inv* = *this*(4)
consider
 $(\text{dpll-end}) \exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i))$
 $| (\text{dpll-more}) \neg (\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i)))$
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 $\text{Min-in}[OF \langle \text{finite } ?I \rangle \langle ?I \neq \{\} \rangle]$ **by** *auto*
moreover have $\forall k < ?i. \text{ learn-or-forget } (f\ k) (f\ (\text{Suc } k))$
using $\text{Min.coboundedI}[of \{i. i \leq i_0 \wedge \neg \text{ learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{ forget}_{NOT} (f\ i) (f\ (\text{Suc } i))\}, \text{simplified}]$
by $(\text{meson } (\neg \text{ learn } (f\ i_0) (f\ (\text{Suc } i_0)) \wedge \neg \text{ forget}_{NOT} (f\ i_0) (f\ (\text{Suc } i_0))) \text{ less-imp-le dual-order.trans not-le})$
ultimately show $?thesis$ **using** *that* **by** *blast*
qed
def $g \equiv \lambda n. f\ (n + \text{Suc } i)$
have $\text{dpll-bj } (f\ i) (g\ 0)$
using $(\neg \text{ learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{ forget}_{NOT} (f\ i) (f\ (\text{Suc } i))) \text{ cdcl}_{NOT} \text{ cdcl}_{NOT}.\text{cases}$
 $g\text{-def}$ **by** *auto*
{
fix j
assume $j \leq i$
then have $\text{learn-or-forget}^{**} (f\ 0) (f\ j)$
apply $(\text{induction } j)$
apply *simp*
by $(\text{metis } (\text{no-types, lifting}) \text{Suc-leD Suc-le-lessD rtranclp.simps } \langle \forall k < i. \text{ learn } (f\ k) (f\ (\text{Suc } k)) \vee \text{ forget}_{NOT} (f\ k) (f\ (\text{Suc } k)) \rangle)$
}
then have $\text{learn-or-forget}^{**} (f\ 0) (f\ i)$ **by** *blast*
then have $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (g\ 0))$
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (f\ 0))$
using $\text{learn-or-forget-dpll-}\mu_C[\text{of } f\ 0\ f\ i\ g\ 0\ A] \text{ inv } \langle \text{dpll-bj } (f\ i) (g\ 0) \rangle$
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 $\text{rtranclp-learn-or-forget-cdcl}_{NOT}[\text{of } f\ 0\ f\ i] \langle \text{learn-or-forget}^{**} (f\ 0) (f\ i) \rangle$
 $\text{cdcl}_{NOT}[\text{of } i]$ **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 $\text{inv cdcl}_{NOT}\text{-}i \text{ rtranclp-cdcl}_{NOT}\text{-trail-clauses-bound } g\text{-def cdcl}_{NOT}\text{-NOT-all-inv}$ **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
{
fix k
assume $k \geq j + \text{Suc } i$
then have $\text{learn-or-forget } (f\ k) (f\ (\text{Suc } k))$

```

    using j[of k-Suc i] unfolding g-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 = this(1) and cdclNOT = this(2) and inv = this(3)
show ?case
proof (rule ccontr)
  assume  $\neg ?case$ 
  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 i0 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 = 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  $\neq \{\}$ ⟩] 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  $\neg \text{learn } (f i_0) (f (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f i_0) (f (\text{Suc } i_0)) \rangle \text{less-imp-le dual-order.trans not-le}$ )
    ultimately show ?thesis using that by blast
  qed
  have dpll-bj (f i) (f (Suc i))
    using  $\neg \text{learn } (f i) (f (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f i) (f (\text{Suc } i)) \rangle \text{cdcl}_{NOT} \text{cdcl}_{NOT}.\text{cases}$ 
    by blast
  {
    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 rtranclp.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 learn-or-forget** (f 0) (f i) by blast

  then show False
    using learn-or-forget-dpll- $\mu_C$ [of f 0 f i f (Suc i) A] inv 0
     $\langle \text{dpll-bj } (f i) (f (\text{Suc } i)) \rangle \text{unfolding cdcl}_{NOT}\text{-NOT-all-inv-def by linarith}$ 
  qed
qed

```

lemma wf-cdcl_{NOT}-no-learn-and-forget-infinite-chain:

assumes
no-infinite-lf: $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$
shows $wf \{(T, S). \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT-}NOT\text{-all-inv } A S\}$ **(is** $wf \{(T, S). \text{cdcl}_{NOT} S T \wedge ?inv S\}$
unfolding *wf-iff-no-infinite-down-chain*
proof (*rule ccontr*)
assume $\neg \neg (\exists f. \forall i. (f (Suc i), f i) \in \{(T, S). \text{cdcl}_{NOT} S T \wedge ?inv S\})$
then obtain *f* **where**
 $\forall i. \text{cdcl}_{NOT} (f i) (f (Suc i)) \wedge ?inv (f i)$
by *fast*
then have $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i))$
using *infinite-cdcl_{NOT}-exists-learn-and-forget-infinite-chain*[*of f*] **by** *meson*
then show *False* **using** *no-infinite-lf* **by** *blast*
qed

lemma *inv-and-tranclp-cdcl_{NOT}-tranclp-cdcl_{NOT}-and-inv*:
 $\text{cdcl}_{NOT}^{++} S T \wedge \text{cdcl}_{NOT-}NOT\text{-all-inv } A S \longleftrightarrow (\lambda S T. \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT-}NOT\text{-all-inv } A S)^{++} S T$
(is $?A \wedge ?I \longleftrightarrow ?B$
proof
assume $?A \wedge ?I$
then have $?A$ **and** $?I$ **by** *blast+*
then show $?B$
apply *induction*
apply (*simp add: tranclp.r-into-trancl*)
by (*metis (no-types, lifting) cdcl_{NOT}-NOT-all-inv tranclp.simps tranclp-into-rtranclp*)
next
assume $?B$
then have $?A$ **by** *induction auto*
moreover have $?I$ **using** $\langle ?B \rangle \text{tranclpD}$ **by** *fastforce*
ultimately show $?A \wedge ?I$ **by** *blast*
qed

lemma *wf-tranclp-cdcl_{NOT}-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$
shows $wf \{(T, S). \text{cdcl}_{NOT}^{++} S T \wedge \text{cdcl}_{NOT-}NOT\text{-all-inv } A S\}$
using *wf-tranclp*[*OF wf-cdcl_{NOT}-no-learn-and-forget-infinite-chain* [*OF no-infinite-lf*]]
apply (*rule wf-subset*)
by (*auto simp: trancl-set-tranclp inv-and-tranclp-cdcl_{NOT}-tranclp-cdcl_{NOT}-and-inv*)

lemma *cdcl_{NOT}-final-state*:
assumes
n-s: *no-step* $\text{cdcl}_{NOT} S$ **and**
inv: $\text{cdcl}_{NOT-}NOT\text{-all-inv } A S$ **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_{NOT}.simps*)
show *?thesis*
apply (*rule dpll-backjump-final-state*[*of S A*])
using *inv decomp n-s'* **unfolding** *cdcl_{NOT}-NOT-all-inv-def* **by** *auto*
qed

lemma *full-cdcl_{NOT}-final-state*:

assumes

full: *full cdcl_{NOT} S T and*

inv: *cdcl_{NOT}-NOT-all-inv A S and*

n-d: *no-dup (trail S) and*

decomp: *all-decomposition-implies-m (clauses 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_{NOT}** S T and n-s*: *no-step cdcl_{NOT} T*

using *full unfolding full-def by blast+*

have *n-s'*: *cdcl_{NOT}-NOT-all-inv A T*

using *cdcl_{NOT}-NOT-all-inv inv st by blast*

moreover have *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*

using *cdcl_{NOT}-NOT-all-inv-def decomp inv rtracp-cdcl_{NOT}-all-decomposition-implies st by auto*

ultimately show *?thesis*

using *cdcl_{NOT}-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-learn* =

conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}

propagate-conds inv backjump-conds

$\lambda C S. \text{distinct-mset } C \wedge \neg \text{tautology } C \wedge \text{learn-restrictions } C S \wedge$

$(\exists F K d F' C' L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge C = C' + \{\#L\} \wedge F \models_{as} C \text{Not } C'$

$\wedge C' + \{\#L\} \notin \text{clauses } S)$

$\lambda C S. \neg (\exists F' F K d L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } (C - \{\#L\}))$

$\wedge \text{forget-restrictions } C S$

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-cl_{NOT} remove-cl_{NOT}:: *'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 clause* \Rightarrow *'v literal* \Rightarrow *'st* \Rightarrow *'st* \Rightarrow *bool and*

learn-restrictions forget-restrictions :: *'v clause* \Rightarrow *'st* \Rightarrow *bool*

begin

lemma *cdcl_{NOT}-learn-all-induct*[*consumes 1*, *case-names dp_{ll}-bj learn forget_{NOT}*]:

fixes *S T* :: *'st*

assumes *cdcl_{NOT} S T and*

dp_{ll}: $\bigwedge T. \text{dp_{ll}-bj } S T \Longrightarrow P S T$ **and**

learning:

$\bigwedge C F K F' C' L T. \text{clauses } S \models_{pm} C$

$\Longrightarrow \text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$

$\Longrightarrow \text{distinct-mset } C \Longrightarrow \neg \text{tautology } C \Longrightarrow \text{learn-restrictions } C S$

$\Longrightarrow \text{trail } S = F' @ \text{Marked } K () \# F \Longrightarrow C = C' + \{\#L\} \Longrightarrow F \models_{as} C \text{Not } C'$

$\Longrightarrow C' + \{\#L\} \notin \text{clauses } S \Longrightarrow T \sim \text{add-cl_{NOT} } C S$

```

     $\Rightarrow P S T$  and
    forgetting:  $\bigwedge C T. \text{ clauses } S - \text{ replicate-mset } (\text{count } (\text{clauses } S) C) C \models_{pm} C$ 
     $\Rightarrow C \in \# \text{ clauses } S$ 
     $\Rightarrow \neg(\exists F' F K L. \text{ trail } S = F' @ \text{ Marked } K () \# F \wedge F \models_{as} CNot (C - \{\#L\# \}))$ 
     $\Rightarrow T \sim \text{ remove-cl}_{NOT} C S$ 
     $\Rightarrow \text{ forget-restrictions } C S \Rightarrow P S T$ 
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  $\Rightarrow$  inv S  $\Rightarrow$  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)
   $\subseteq \text{ build-all-simple-clss } (\text{atms-of-msu } (\text{clauses } S) \cup \text{ atm-of ' lits-of (trail S)})$ 
proof
fix  $C$  assume  $C: C \in \text{ set-mset } (\text{clauses } T - \text{ clauses } S)$ 
have distinct-mset C  $\neg$ tautology C using learn C n-d by (elim learnE; auto)+
then have  $C \in \text{ build-all-simple-clss } (\text{atms-of } C)$ 
  using distinct-mset-not-tautology-implies-in-build-all-simple-clss by blast
moreover have  $\text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of (trail S)}$ 
  using learn C n-d by (elim learnE) (auto simp: atms-of-ms-def atms-of-def image-Un
true-annots-CNot-all-atms-defined)
moreover have finite (atms-of-msu (clauses S)  $\cup$  atm-of ' lits-of (trail S))
  by auto
ultimately show  $C \in \text{ build-all-simple-clss } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of (trail S)})$ 
  using build-all-simple-clss-mono by (metis (no-types) insert-subset mk-disjoint-insert)
qed

definition conflicting-bj-clss S  $\equiv$ 
   $\{C + \{\#L\#\} \mid C L. C + \{\#L\#\} \in \# \text{ clauses } S \wedge \text{ distinct-mset } (C + \{\#L\#\}) \wedge \neg \text{tautology } (C + \{\#L\#\})$ 
 $\wedge (\exists F' K F. \text{ trail } S = F' @ \text{ Marked } K () \# F \wedge F \models_{as} CNot C)\}$ 

lemma conflicting-bj-clss-remove-clNOT[simp]:
  conflicting-bj-clss (remove-clNOT C S) = conflicting-bj-clss S - {C}
unfolding conflicting-bj-clss-def by fastforce

lemma conflicting-bj-clss-add-clNOT-state-eq:
   $T \sim \text{ add-cl}_{NOT} C' S \Rightarrow \text{ no-dup } (\text{trail } S) \Rightarrow \text{ conflicting-bj-clss } T$ 
 $= \text{ conflicting-bj-clss } S$ 
 $\cup (\text{ if } \exists C L. C' = C + \{\#L\#\} \wedge \text{ distinct-mset } (C + \{\#L\#\}) \wedge \neg \text{tautology } (C + \{\#L\#\})$ 
 $\wedge (\exists F' K d F. \text{ trail } S = F' @ \text{ Marked } K () \# F \wedge F \models_{as} CNot C)$ 
 $\text{ then } \{C'\} \text{ else } \{\})$ 

```

unfolding *conflicting-bj-clss-def* **by** *auto metis+*

lemma *conflicting-bj-clss-add-clss_{NOT}*:

no-dup (*trail S*) \implies
conflicting-bj-clss (*add-clss_{NOT} C' S*)
 $=$ *conflicting-bj-clss S*
 \cup (*if* $\exists C L. C' = C + \{\#L\} \wedge \text{distinct-mset } (C + \{\#L\}) \wedge \neg \text{tautology } (C + \{\#L\})$
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} C \text{Not } C)$
then $\{C'\}$ *else* $\{\}$)
using *conflicting-bj-clss-add-clss_{NOT}-state-eq* **by** *auto*

lemma *conflicting-bj-clss-incl-clauses*:

conflicting-bj-clss S \subseteq *set-mset (clauses S)*
unfolding *conflicting-bj-clss-def* **by** *auto*

lemma *finite-conflicting-bj-clss[simp]*:

finite (conflicting-bj-clss S)
using *conflicting-bj-clss-incl-clauses[of S]* *rev-finite-subset* **by** *blast*

lemma *learn-conflicting-increasing*:

no-dup (*trail S*) \implies *learn S T* \implies *conflicting-bj-clss S* \subseteq *conflicting-bj-clss T*
apply (*elim learnE*)
by (*subst conflicting-bj-clss-add-clss_{NOT}-state-eq[of T]*) *auto*

abbreviation *conflicting-bj-clss-yet b S* \equiv

$\exists \wedge b - \text{card } (\text{conflicting-bj-clss } S)$

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 *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*:

assumes *forget_{NOT} S T*
shows *conflicting-bj-clss S* $=$ *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_{NOT}*
diff-union-cancelR insert-iff mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1)
state-eq_{NOT}-clauses state-eq_{NOT}-trail trail-remove-clss_{NOT})

lemma *forget- μ_L -decrease*:

assumes *forget_{NOT}: forget_{NOT} S T*
shows $(\mu_L b T, \mu_L b S) \in \text{less-than} <*\text{lex}*> \text{less-than}$

proof –

have *card (set-mset (clauses T))* $<$ *card (set-mset (clauses S))*
using *forget_{NOT} apply induction*
by (*metis card-Diff1-less clauses-remove-clss_{NOT} finite-set-mset mem-set-mset-iff order-refl*
set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
then show *?thesis*
unfolding *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched[OF forget_{NOT}]*
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- μ_L -decrease*:

assumes *learnST*: *learn* $S \ T$ **and** *n-d*: *no-dup* (*trail* S) **and**

A : *atms-of-msu* (*clauses* S) \cup *atm-of* ' *lits-of* (*trail* S) $\subseteq A$ **and**

fin-A: *finite* A

shows $(\mu_L (\text{card } A) \ T, \mu_L (\text{card } A) \ S) \in \text{less-than } <*\text{lex}*> \text{less-than}$

proof –

have [*simp*]: (*atms-of-msu* (*clauses* T) \cup *atm-of* ' *lits-of* (*trail* T))

= (*atms-of-msu* (*clauses* S) \cup *atm-of* ' *lits-of* (*trail* S))

using *learnST* *n-d* **by** (*elim learnE*) *auto*

then have *card* (*atms-of-msu* (*clauses* T) \cup *atm-of* ' *lits-of* (*trail* T))

= *card* (*atms-of-msu* (*clauses* S) \cup *atm-of* ' *lits-of* (*trail* S))

by (*auto intro!*: *card-mono*)

then have 3 : $(3::\text{nat}) \wedge \text{card} (\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of ' lits-of} (\text{trail } T))$

= $3 \wedge \text{card} (\text{atms-of-msu} (\text{clauses } S) \cup \text{atm-of ' lits-of} (\text{trail } S))$

by (*auto intro*: *power-mono*)

moreover have *conflicting-bj-clss* $S \subseteq \text{conflicting-bj-clss } T$

using *learnST* *n-d* **by** (*simp add*: *learn-conflicting-increasing*)

moreover have *conflicting-bj-clss* $S \neq \text{conflicting-bj-clss } T$

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** $T = \text{this}(4)$

then obtain $F \ K \ F' \ C' \ L$ **where**

tr-S: *trail* $S = F' @ \text{Marked } K \ () \ \# \ F$ **and**

C : $C = C' + \{\#L\# \}$ **and**

F : $F \models_{\text{as}} C \text{Not } C'$ **and**

$C-S$: $C' + \{\#L\# \} \notin \text{clauses } S$

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 $T \ n-d$ **unfolding** *conflicting-bj-clss-def* **by** *fastforce*

moreover have $C' + \{\#L\# \} \notin \text{conflicting-bj-clss } S$

using $C-S$ **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-clss_NOT*)

ultimately have *card* (*conflicting-bj-clss* T) $\geq \text{card} (\text{conflicting-bj-clss } S)$

using *card-mono* **by** *blast*

moreover

have *fin'*: *finite* (*atms-of-msu* (*clauses* T) \cup *atm-of* ' *lits-of* (*trail* T))

by *auto*

have 1 : *atms-of-ms* (*conflicting-bj-clss* T) $\subseteq \text{atms-of-msu} (\text{clauses } T)$

unfolding *conflicting-bj-clss-def* *atms-of-ms-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

$\subseteq \text{build-all-simple-clss} (\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of ' lits-of} (\text{trail } T))$

by *standard* (*meson* $1 \ 2 \ \text{fin}' \ \langle \text{finite} (\text{conflicting-bj-clss } T) \rangle \text{build-all-simple-clss-mono}$)

distinct-mset-set-def simplified-in-build-all subsetCE sup.coboundedI1)
moreover
then have $\# : 3 \wedge \text{card} (\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of } ' \text{lits-of } (\text{trail } T))$
 $\geq \text{card} (\text{conflicting-bj-clss } T)$
by (*meson Nat.le-trans build-all-simple-clss-card build-all-simple-clss-finite card-mono fin'*)
have $\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of } ' \text{lits-of } (\text{trail } T) \subseteq A$
using *learnE[OF learnST] A by simp*
then have $3 \wedge (\text{card } A) \geq \text{card} (\text{conflicting-bj-clss } T)$
using $\# \text{ fin-A by } (\text{meson build-all-simple-clss-card build-all-simple-clss-finite}$
 $\text{build-all-simple-clss-mono calculation(2) card-mono dual-order.trans})$
ultimately show *?thesis*
using *psubset-card-mono[OF fin-T]*
unfolding *less-than-iff lex-prod-def by clarify*
 $(\text{meson } \langle \text{conflicting-bj-clss } S \neq \text{conflicting-bj-clss } T \rangle$
 $\langle \text{conflicting-bj-clss } S \subseteq \text{conflicting-bj-clss } T \rangle$
 $\text{diff-less-mono2 le-less-trans not-le psubsetI})$
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 $\text{atm-of } ' \text{lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ and in the clauses $\text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A$. This can be the set of all the literals in the starting set of clauses.
- *no-dup (trail S)*: no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} **where**

$\mu_{CDCL} A T \equiv ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T),$
 $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T, \text{card} (\text{set-mset} (\text{clauses } T)))$

lemma *cdcl_{NOT}-decreasing-measure:*

assumes
 $\text{cdcl}_{NOT} S T$ **and**
 $\text{inv: inv } S$ **and**
 $\text{atm-clss: atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-lits: atm-of } ' \text{lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$ **and**
 $\text{fin-A: finite } A$
shows $(\mu_{CDCL} A T, \mu_{CDCL} A S)$
 $\in \text{less-than } <*\text{lex}*> (\text{less-than } <*\text{lex}*> \text{less-than})$
using *assms(1)*
proof *induction*
case $(c\text{-dpll-bj } T)$
from *dpll-bj-trail-mes-decreasing-prop[OF this(1) inv atm-clss atm-lits n-d fin-A]*
show *?case unfolding $\mu_{CDCL}\text{-def}$*
 $\text{by } (\text{meson in-lex-prod less-than-iff})$
next
case $(c\text{-learn } T)$ **note** $\text{learn} = \text{this}(1)$
then have $S: \text{trail } S = \text{trail } T$
using $\text{inv atm-clss atm-lits n-d fin-A}$
by *(elim learnE) auto*
show *?case*
using *learn- μ_L -decrease[OF learn -] atm-clss atm-lits fin-A n-d unfolding S $\mu_{CDCL}\text{-def}$ by auto*

next

case (*c-forget*_{NOT} *T*) **note** *forget*_{NOT} = *this*(1)
have *trail S* = *trail T* **using** *forget*_{NOT} **by** *induction auto*
then show ?*case*
 using *forget-μ_L-decrease*[*OF forget*_{NOT}] **unfolding** *μ_{CDCL}-def* **by** *auto*
qed

lemma *wf-cdcl_{NOT}-restricted-learning*:

assumes *finite A*
shows *wf* {(*T*, *S*).
 (*atms-of-msu* (*clauses S*) ⊆ *atms-of-ms A* ∧ *atm-of* ‘*lits-of* (*trail S*) ⊆ *atms-of-ms A*
 ∧ *no-dup* (*trail S*)
 ∧ *inv S*)
 ∧ *cdcl_{NOT} S T* }
by (*rule wf-wf-if-measure'*[*of less-than <*lex*> (less-than <*lex*> less-than)*])
 (*auto intro: cdcl_{NOT}-decreasing-measure*[*OF - - - - assms*])

definition *μ_{C'}* :: '*v literal multiset set* ⇒ '*st* ⇒ *nat* **where**

μ_{C'} A T ≡ *μ_C* (1+*card* (*atms-of-ms A*)) (2+*card* (*atms-of-ms A*)) (*trail-weight T*)

definition *μ_{CDCL'}* :: '*v literal multiset set* ⇒ '*st* ⇒ *nat* **where**

μ_{CDCL'} A T ≡
 ((2+*card* (*atms-of-ms A*)) ^ (1+*card* (*atms-of-ms A*)) - *μ_{C'} A T*) * (1 + 3^*card* (*atms-of-ms A*)) *
 2
 + *conflicting-bj-clss-yet* (*card* (*atms-of-ms A*)) *T* * 2
 + *card* (*set-mset* (*clauses T*))

lemma *cdcl_{NOT}-decreasing-measure'*:

assumes
 cdcl_{NOT} S T **and**
 inv: inv S **and**
 atms-clss: atms-of-msu (*clauses S*) ⊆ *atms-of-ms A* **and**
 atms-trail: atm-of ‘*lits-of* (*trail S*) ⊆ *atms-of-ms A* **and**
 n-d: no-dup (*trail S*) **and**
 fin-A: finite A

shows *μ_{CDCL'} A T* < *μ_{CDCL'} A S*

using *assms*(1)

proof (*induction rule: cdcl_{NOT}-learn-all-induct*)

case (*dpll-bj T*)

then have (2+*card* (*atms-of-ms A*)) ^ (1+*card* (*atms-of-ms A*)) - *μ_{C'} A T*

< (2+*card* (*atms-of-ms A*)) ^ (1+*card* (*atms-of-ms A*)) - *μ_{C'} A S*

using *dpll-bj-trail-mes-decreasing-prop fin-A inv n-d atms-clss atms-trail*

unfolding *μ_{C'}-def* **by** *blast*

then have *XX*: ((2+*card* (*atms-of-ms A*)) ^ (1+*card* (*atms-of-ms A*)) - *μ_{C'} A T*) + 1

≤ (2+*card* (*atms-of-ms A*)) ^ (1+*card* (*atms-of-ms A*)) - *μ_{C'} A S*

by *auto*

from *mult-le-mono1*[*OF this, of (1 + 3 ^ card* (*atms-of-ms A*))]

have ((2 + *card* (*atms-of-ms A*)) ^ (1 + *card* (*atms-of-ms A*)) - *μ_{C'} A T*) *

(1 + 3 ^ *card* (*atms-of-ms A*)) + (1 + 3 ^ *card* (*atms-of-ms A*))

≤ ((2 + *card* (*atms-of-ms A*)) ^ (1 + *card* (*atms-of-ms A*)) - *μ_{C'} A S*)

* (1 + 3 ^ *card* (*atms-of-ms A*))

unfolding *Nat.add-mult-distrib*

by *presburger*

moreover

have *cl-T-S: clauses T* = *clauses S*

```

    using dpll-bj.hyps inv dpll-bj-clauses by auto
    have conflicting-bj-clss-yet (card (atms-of-ms A))  $S < 1 + 3 \wedge \text{card (atms-of-ms A)}$ 
    by simp
    ultimately have  $((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T)$ 
       $* (1 + 3 \wedge \text{card (atms-of-ms A)}) + \text{conflicting-bj-clss-yet (card (atms-of-ms A)) T}$ 
       $< ((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A S) * (1 + 3 \wedge \text{card (atms-of-ms A)})$ 
    A))
    by linarith
    then have  $((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T)$ 
       $* (1 + 3 \wedge \text{card (atms-of-ms A)})$ 
       $+ \text{conflicting-bj-clss-yet (card (atms-of-ms A)) T}$ 
       $< ((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A S)$ 
       $* (1 + 3 \wedge \text{card (atms-of-ms A)})$ 
       $+ \text{conflicting-bj-clss-yet (card (atms-of-ms A)) S}$ 
    by linarith
    then have  $((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T)$ 
       $* (1 + 3 \wedge \text{card (atms-of-ms A)}) * 2$ 
       $+ \text{conflicting-bj-clss-yet (card (atms-of-ms A)) T} * 2$ 
       $< ((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A S)$ 
       $* (1 + 3 \wedge \text{card (atms-of-ms A)}) * 2$ 
       $+ \text{conflicting-bj-clss-yet (card (atms-of-ms A)) S} * 2$ 
    by linarith
    then show ?case unfolding  $\mu_{CDCL}'\text{-def cl-T-S}$  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' = this(7) and
  F-C = this(8) and C-new = this(9) and T = this(10)
have insert C (conflicting-bj-clss S)  $\subseteq \text{build-all-simple-clss (atms-of-ms A)}$ 
proof -
  have  $C \in \text{build-all-simple-clss (atms-of-ms A)}$ 
  by (metis (no-types, hide-lams) Un-subset-iff atms-of-ms-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  $\subseteq \text{build-all-simple-clss (atms-of-ms A)}$ 
  unfolding conflicting-bj-clss-def
  proof
    fix x :: 'v literal multiset
    assume  $x \in \{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \text{ clauses } S$ 
       $\wedge \text{distinct-mset (C + \{\#L\# \})} \wedge \neg \text{tautology (C + \{\#L\# \})}$ 
       $\wedge (\exists F' K F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} C \text{Not } C)\}$ 
    then have  $\exists m l. x = m + \{\#l\# \} \wedge m + \{\#l\# \} \in \# \text{ clauses } S$ 
       $\wedge \text{distinct-mset (m + \{\#l\# \})} \wedge \neg \text{tautology (m + \{\#l\# \})}$ 
       $\wedge (\exists ms l msa. \text{trail } S = ms @ \text{Marked } l () \# msa \wedge msa \models_{\text{as}} C \text{Not } m)$ 
    by blast
    then show  $x \in \text{build-all-simple-clss (atms-of-ms A)}$ 
    by (meson atms-clss atms-of-atms-of-ms-mono atms-of-ms-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  $\text{card (insert C (conflicting-bj-clss S))} \leq 3 \wedge (\text{card (atms-of-ms A)})$ 
  by (meson Nat.le-trans atms-of-ms-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 $(metis \text{ (no-types) } C' \text{ C-new card-insert-if conflicting-bj-clss-incl-clauses contra-subsetD}$
 $\text{finite-conflicting-bj-clss mem-set-mset-iff})$
moreover have $[simp]: \text{conflicting-bj-clss } (\text{add-cl}_{NOT} C S) = \text{conflicting-bj-clss } S \cup \{C\}$
using $\text{dist tauto } F\text{-}C \text{ n-d by } (\text{subst conflicting-bj-clss-add-cl}_{NOT})$
 $(\text{force simp add: ac-simps } C' \text{ tr-}S)+$
ultimately have $[simp]: \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) S$
 $= \text{Suc } (\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S))$
by simp
have 1: $\text{clauses } T = \text{clauses } (\text{add-cl}_{NOT} C S)$ **using** T **by auto**
have 2: $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T$
 $= \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S)$
using T **unfolding** $\text{conflicting-bj-clss-def}$ **by auto**
have 3: $\mu_{C'} A T = \mu_{C'} A (\text{add-cl}_{NOT} C S)$
using T **unfolding** $\mu_{C'}\text{-def}$ **by auto**
have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_{C'} A (\text{add-cl}_{NOT} C S))$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $= ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_{C'} A S)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
using $\text{n-d unfolding } \mu_{C'}\text{-def}$ **by auto**
moreover
have $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S)$
 $* 2$
 $+ \text{card } (\text{set-mset } (\text{clauses } (\text{add-cl}_{NOT} C S)))$
 $< \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) S * 2$
 $+ \text{card } (\text{set-mset } (\text{clauses } S))$
by $(\text{simp add: } C' \text{ C-new n-d})$
ultimately show $?case$ **unfolding** $\mu_{CDCL}'\text{-def 1 2 3}$ **by presburger**
next
case $(\text{forget}_{NOT} C T)$ **note** $T = \text{this}(4)$
have $[simp]: \mu_{C'} A (\text{remove-cl}_{NOT} C S) = \mu_{C'} A S$
unfolding $\mu_{C'}\text{-def}$ **by auto**
have $\text{forget}_{NOT} S T$
apply $(\text{rule forget}_{NOT}.\text{intros})$ **using** forget_{NOT} **by 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 blast**
moreover have $\text{card } (\text{set-mset } (\text{clauses } T)) < \text{card } (\text{set-mset } (\text{clauses } S))$
by $(metis T \text{ card-Diff1-less clauses-remove-cl}_{NOT} \text{ finite-set-mset forget}_{NOT}.\text{hyps}(2)$
 $\text{mem-set-mset-iff order-refl set-mset-minus-replicate-mset}(1) \text{ state-eq}_{NOT}\text{-clauses})$
ultimately show $?case$ **unfolding** $\mu_{CDCL}'\text{-def}$
by $(metis \text{ (no-types) } T \langle \mu_{C'} A (\text{remove-cl}_{NOT} C S) = \mu_{C'} A S \rangle \text{ add-le-cancel-left}$
 $\mu_{C'}\text{-def not-le state-eq}_{NOT}\text{-trail})$
qed

lemma $\text{cdcl}_{NOT}\text{-clauses-bound}$:

assumes

$\text{cdcl}_{NOT} S T$ **and**

$\text{inv } S$ **and**

$\text{atms-of-msu } (\text{clauses } S) \subseteq A$ **and**

$\text{atm-of } (\text{lits-of } (\text{trail } S)) \subseteq A$ **and**

$\text{n-d: no-dup } (\text{trail } S)$ **and**

$\text{fin-A}[simp]: \text{finite } A$

shows $\text{set-mset } (\text{clauses } T) \subseteq \text{set-mset } (\text{clauses } S) \cup \text{build-all-simple-clss } A$

using assms

proof (*induction rule: cdcl_{NOT}-learn-all-induct*)
 case *dpll-bj*
 then show ?case using *dpll-bj-clauses* by *simp*
 next
 case *forget_{NOT}*
 then show ?case using *clauses-remove-cl_{NOT}* unfolding *state-eq_{NOT}-def* by *auto*
 next
 case (*learn C F K d F' C' L*) note *atms-C = this(2)* and *dist = this(3)* and *tauto = this(4)* and
T = this(10) and *atms-clss-S = this(12)* and *atms-trail-S = this(13)*
 have *atms-of C ⊆ A*
 using *atms-C atms-clss-S atms-trail-S* by *auto*
 then have *build-all-simple-clss (atms-of C) ⊆ build-all-simple-clss A*
 by (*simp add: build-all-simple-clss-mono*)
 then have *C ∈ build-all-simple-clss A*
 using *finite dist tauto*
 by (*auto dest: distinct-mset-not-tautology-implies-in-build-all-simple-clss*)
 then show ?case using *T n-d* by *auto*
 qed

lemma *rtrancpl-cdcl_{NOT}-clauses-bound:*

assumes
*cdcl_{NOT}** S T* and
inv S and
atms-of-msu (clauses S) ⊆ A and
atm-of '(lits-of (trail S)) ⊆ A and
n-d: no-dup (trail S) and
finite: finite A
 shows *set-mset (clauses T) ⊆ set-mset (clauses S) ∪ build-all-simple-clss A*
 using *assms(1-5)*
proof *induction*
 case *base*
 then show ?case by *simp*
 next
 case (*step T U*) note *st = this(1)* and *cdcl_{NOT} = 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-cl_S = this(7)*
 have *inv T*
 using *rtrancpl-cdcl_{NOT}-inv st inv* by *blast*
 moreover have *atms-of-msu (clauses T) ⊆ A* and *atm-of '(lits-of (trail T)) ⊆ 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 (inv S) n-d]* by *simp*
 ultimately have *set-mset (clauses U) ⊆ set-mset (clauses T) ∪ build-all-simple-clss A*
 using *cdcl_{NOT} finite n-d* by (*auto simp: cdcl_{NOT}-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-msu (clauses S) ⊆ A and
atm-of '(lits-of (trail S)) ⊆ A and
n-d: no-dup (trail S) and
finite: finite A

shows $\text{card } (\text{set-mset } (\text{clauses } T)) \leq \text{card } (\text{set-mset } (\text{clauses } S)) + 3 \wedge (\text{card } A)$
using $\text{rtrancpl-cdcl}_{NOT}\text{-clauses-bound}[OF \text{ assms}] \text{ finite}$ **by** $(\text{meson } \text{Nat.le-trans}$
 $\text{build-all-simple-clss-card } \text{build-all-simple-clss-finite } \text{card-Un-le } \text{card-mono } \text{finite-UnI}$
 $\text{finite-set-mset } \text{nat-add-left-cancel-le})$

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-card-clauses-bound}'$:

assumes
 $\text{cdcl}_{NOT}^{**} S T$ **and**
 $\text{inv } S$ **and**
 $\text{atms-of-msu } (\text{clauses } S) \subseteq A$ **and**
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq A$ **and**
 $\text{n-d: no-dup } (\text{trail } S)$ **and**
 $\text{finite: finite } A$
shows $\text{card } \{C \mid C. C \in \# \text{ clauses } T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$
 $\leq \text{card } \{C \mid C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } A)$
 $(\text{is card } ?T \leq \text{card } ?S + -)$

using $\text{rtrancpl-cdcl}_{NOT}\text{-clauses-bound}[OF \text{ assms}] \text{ finite}$

proof –

have $?T \subseteq ?S \cup \text{build-all-simple-clss } A$
using $\text{rtrancpl-cdcl}_{NOT}\text{-clauses-bound}[OF \text{ assms}]$ **by** force
then have $\text{card } ?T \leq \text{card } (?S \cup \text{build-all-simple-clss } A)$
using finite **by** $(\text{simp add: assms}(5) \text{ build-all-simple-clss-finite } \text{card-mono})$
then show $?thesis$
by $(\text{meson } \text{le-trans } \text{build-all-simple-clss-card } \text{card-Un-le } \text{local.finite } \text{nat-add-left-cancel-le})$

qed

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-card-simple-clauses-bound}$:

assumes
 $\text{cdcl}_{NOT}^{**} S T$ **and**
 $\text{inv } S$ **and**
 $\text{atms-of-msu } (\text{clauses } S) \subseteq A$ **and**
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq A$ **and**
 $\text{n-d: no-dup } (\text{trail } S)$ **and**
 $\text{finite: finite } A$
shows $\text{card } (\text{set-mset } (\text{clauses } T))$
 $\leq \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } A)$
 $(\text{is card } ?T \leq \text{card } ?S + -)$

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** $(\text{metis } (\text{no-types, hide-lams}) \text{Un-iff } \text{assms}(3)$
 $\text{atms-of-atms-of-ms-mono } \text{build-all-simple-clss-mono } \text{contra-subsetD}$
 $\text{distinct-mset-not-tautology-implies-in-build-all-simple-clss } \text{local.finite } \text{mem-set-mset-iff}$
 $\text{subset-trans})$
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** $(\text{simp add: assms}(5) \text{ build-all-simple-clss-finite } \text{card-mono})$
then show $?thesis$
by $(\text{meson } \text{le-trans } \text{build-all-simple-clss-card } \text{card-Un-le } \text{local.finite } \text{nat-add-left-cancel-le})$

qed

definition $\mu_{CDCL}'\text{-bound} :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_{CDCL}'\text{-bound } A S =$
 $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$

$+ 2 * 3 \wedge (\text{card } (\text{atms-of-ms } A))$
 $+ \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } (\text{atms-of-ms } A))$

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 $rtrancpl\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}:$

assumes

$cdcl_{NOT}^{**} S T$ **and**

$inv S$ **and**

$\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } (\text{atms-of-ms } A)$ **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 $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A U)$

$\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$

by *auto*

then have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A U)$

$* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$

$\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$

using *mult-le-mono1* **by** *blast*

moreover

have $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T * 2 \leq 2 * 3 \wedge \text{card } (\text{atms-of-ms } A)$

by *linarith*

moreover have $\text{card } (\text{set-mset } (\text{clauses } U))$

$\leq \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge \text{card } (\text{atms-of-ms } A)$

using $rtrancpl\text{-}cdcl_{NOT}\text{-card-simple-clauses-bound}[OF \text{ assms}(1-6)] U$ **by** *auto*

ultimately show *?thesis*

unfolding $\mu_{CDCL}'\text{-def}$ $\mu_{CDCL}'\text{-bound-def}$ **by** *linarith*

qed

lemma $rtrancpl\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-bound}:$

assumes

$cdcl_{NOT}^{**} S T$ **and**

$inv S$ **and**

$\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } (\text{atms-of-ms } A)$

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 $\mu_{CDCL}'\text{-def}$ $\mu_C'\text{-def}$ $\text{conflicting-bj-clss-def}$ **by** *auto*

then show *?thesis* **using** $rtrancpl\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}[OF \text{ assms, of - trail } T]$

$\text{state-eq}_{NOT}\text{-ref}$ **by** *fastforce*

qed

lemma $rtrancpl\text{-}\mu_{CDCL}'\text{-bound-decreasing}:$

assumes

$cdcl_{NOT}^{**} S T$ **and**

$inv S$ **and**

$atms\text{-}of\text{-}msu\ (clauses\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $atm\text{-}of\ ('lits\text{-}of\ (trail\ S)) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ **and**
 $finite[simp]: finite\ (atms\text{-}of\text{-}ms\ A)$
shows $\mu_{CDCL}'\text{-}bound\ A\ T \leq \mu_{CDCL}'\text{-}bound\ A\ S$
proof –
have $\{C. C \in \# \text{ clauses } T \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\}$
 $\subseteq \{C. C \in \# \text{ clauses } S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\}$ (**is** $?T \subseteq ?S$)
proof (*rule Set.subsetI*)
fix C **assume** $C \in ?T$
then have $C\text{-}T: C \in \# \text{ clauses } T$ **and** $t\text{-}d: tautology\ C \vee \neg distinct\text{-}mset\ C$
by *auto*
then have $C \notin build\text{-}all\text{-}simple\text{-}clss\ (atms\text{-}of\text{-}ms\ A)$
by (*auto dest: build-all-simple-clssE*)
then show $C \in ?S$
using $C\text{-}T\ rtrancp\text{-}cdcl_{NOT}\text{-}clauses\text{-}bound[OF\ assms]\ t\text{-}d$ **by** *force*
qed
then have $card\ \{C. C \in \# \text{ clauses } T \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} \leq$
 $card\ \{C. C \in \# \text{ clauses } S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\}$
by (*simp add: card-mono*)
then show $?thesis$
unfolding $\mu_{CDCL}'\text{-}bound\text{-}def$ **by** *auto*
qed
end — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learn*

14.7 CDCL with restarts

14.7.1 Definition

locale *restart-ops* =
fixes
 $cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$ **and**
 $restart :: 'st \Rightarrow 'st \Rightarrow bool$
begin
inductive $cdcl_{NOT}\text{-}raw\text{-}restart :: 'st \Rightarrow 'st \Rightarrow bool$ **where**
 $cdcl_{NOT}\ S\ T \Longrightarrow cdcl_{NOT}\text{-}raw\text{-}restart\ S\ T \mid$
 $restart\ S\ T \Longrightarrow cdcl_{NOT}\text{-}raw\text{-}restart\ S\ T$
end

locale *conflict-driven-clause-learning-with-restarts* =
 $conflict\text{-}driven\text{-}clause\text{-}learning\ trail\ clauses\ prepend\text{-}trail\ tl\text{-}trail\ add\text{-}cls_{NOT}\ remove\text{-}cls_{NOT}$
 $propagate\text{-}conds\ inv\ backjump\text{-}conds\ learn\text{-}cond\ forget\text{-}cond$
for
 $trail :: 'st \Rightarrow ('v, unit, unit)\ marked\text{-}lits$ **and**
 $clauses :: 'st \Rightarrow 'v\ clauses$ **and**
 $prepend\text{-}trail :: ('v, unit, unit)\ marked\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
 $tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
 $add\text{-}cls_{NOT}\ remove\text{-}cls_{NOT} :: 'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
 $propagate\text{-}conds :: ('v, unit, unit)\ marked\text{-}lit \Rightarrow 'st \Rightarrow bool$ **and**
 $inv :: 'st \Rightarrow bool$ **and**
 $backjump\text{-}conds :: 'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **and**
 $learn\text{-}cond\ forget\text{-}cond :: 'v\ clause \Rightarrow 'st \Rightarrow bool$
begin

```

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
  cdclNOT S T  $\longleftrightarrow$  restart-ops.cdclNOT-raw-restart cdclNOT ( $\lambda$ - . False) S T
  (is ?C S T  $\longleftrightarrow$  ?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  $\implies$  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_{NOT}* here):

- a function f that is strictly monotonic. The first step is actually only used as a restart to clean the state (e.g. to ensure that the trail is empty). Then we assume that $(1::'a) \leq f$ n for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...
- a measure μ : it should decrease under the assumptions *bound-inv*, whenever a *cdcl_{NOT}* or a *restart* is done. A parameter is given to μ : for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.
- we also assume that the measure decrease after any *cdcl_{NOT}* step.
- an invariant on the states *cdcl_{NOT}-inv* that also holds after restarts.
- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ -*bound* taking the same parameter as μ and the initial state of the considered *cdcl_{NOT}* chain.

```

locale cdclNOT-increasing-restarts-ops =
  restart-ops cdclNOT restart for
    restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
    cdclNOT :: '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
  cdclNOT-inv :: 'st  $\Rightarrow$  bool and
   $\mu$ -bound :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat
assumes
  f: unbounded f and

```

$f\text{-ge-1}:\bigwedge n. n \geq 1 \implies f\ n \neq 0$ **and**
 $\text{bound-inv}:\bigwedge A\ S\ T. \text{cdcl}_{\text{NOT-inv}}\ S \implies \text{bound-inv}\ A\ S \implies \text{cdcl}_{\text{NOT}}\ S\ T \implies \text{bound-inv}\ A\ T$ **and**
 $\text{cdcl}_{\text{NOT-measure}}:\bigwedge A\ S\ T. \text{cdcl}_{\text{NOT-inv}}\ S \implies \text{bound-inv}\ A\ S \implies \text{cdcl}_{\text{NOT}}\ S\ T \implies \mu\ A\ T < \mu$
 $A\ S$ **and**
 $\text{measure-bound2}:\bigwedge A\ T\ U. \text{cdcl}_{\text{NOT-inv}}\ T \implies \text{bound-inv}\ A\ T \implies \text{cdcl}_{\text{NOT}}^{**}\ T\ U$
 $\implies \mu\ A\ U \leq \mu\text{-bound}\ A\ T$ **and**
 $\text{measure-bound4}:\bigwedge A\ T\ U. \text{cdcl}_{\text{NOT-inv}}\ T \implies \text{bound-inv}\ A\ T \implies \text{cdcl}_{\text{NOT}}^{**}\ T\ U$
 $\implies \mu\text{-bound}\ A\ U \leq \mu\text{-bound}\ A\ T$ **and**
 $\text{cdcl}_{\text{NOT-restart-inv}}:\bigwedge A\ U\ V. \text{cdcl}_{\text{NOT-inv}}\ U \implies \text{restart}\ U\ V \implies \text{bound-inv}\ A\ U \implies \text{bound-inv}$
 $A\ V$
and
 $\text{exists-bound}:\bigwedge R\ S. \text{cdcl}_{\text{NOT-inv}}\ R \implies \text{restart}\ R\ S \implies \exists A. \text{bound-inv}\ A\ S$ **and**
 $\text{cdcl}_{\text{NOT-inv}}:\bigwedge S\ T. \text{cdcl}_{\text{NOT-inv}}\ S \implies \text{cdcl}_{\text{NOT}}\ S\ T \implies \text{cdcl}_{\text{NOT-inv}}\ T$ **and**
 $\text{cdcl}_{\text{NOT-inv-restart}}:\bigwedge S\ T. \text{cdcl}_{\text{NOT-inv}}\ S \implies \text{restart}\ S\ T \implies \text{cdcl}_{\text{NOT-inv}}\ T$
begin

lemma $\text{cdcl}_{\text{NOT-cdcl}_{\text{NOT-inv}}}$:
assumes
 $(\text{cdcl}_{\text{NOT}} \rightsquigarrow n)\ S\ T$ **and**
 $\text{cdcl}_{\text{NOT-inv}}\ S$
shows $\text{cdcl}_{\text{NOT-inv}}\ T$
using *assms* **by** (*induction* n *arbitrary*: T) (*auto intro*: $\text{bound-inv}\ \text{cdcl}_{\text{NOT-inv}}$)

lemma $\text{cdcl}_{\text{NOT-bound-inv}}$:
assumes
 $(\text{cdcl}_{\text{NOT}} \rightsquigarrow n)\ S\ T$ **and**
 $\text{cdcl}_{\text{NOT-inv}}\ S$
 $\text{bound-inv}\ A\ S$
shows $\text{bound-inv}\ A\ T$
using *assms* **by** (*induction* n *arbitrary*: T) (*auto intro*: $\text{bound-inv}\ \text{cdcl}_{\text{NOT-cdcl}_{\text{NOT-inv}}}$)

lemma $\text{rtrancpl-cdcl}_{\text{NOT-cdcl}_{\text{NOT-inv}}}$:
assumes
 $\text{cdcl}_{\text{NOT}}^{**}\ S\ T$ **and**
 $\text{cdcl}_{\text{NOT-inv}}\ S$
shows $\text{cdcl}_{\text{NOT-inv}}\ T$
using *assms* **by** *induction* (*auto intro*: $\text{cdcl}_{\text{NOT-inv}}$)

lemma $\text{rtrancpl-cdcl}_{\text{NOT-bound-inv}}$:
assumes
 $\text{cdcl}_{\text{NOT}}^{**}\ S\ T$ **and**
 $\text{bound-inv}\ A\ S$ **and**
 $\text{cdcl}_{\text{NOT-inv}}\ S$
shows $\text{bound-inv}\ A\ T$
using *assms* **by** *induction* (*auto intro*: $\text{bound-inv}\ \text{rtrancpl-cdcl}_{\text{NOT-cdcl}_{\text{NOT-inv}}}$)

lemma $\text{cdcl}_{\text{NOT-comp-n-le}}$:
assumes
 $(\text{cdcl}_{\text{NOT}} \rightsquigarrow (\text{Suc}\ n))\ S\ T$ **and**
 $\text{bound-inv}\ A\ S$
 $\text{cdcl}_{\text{NOT-inv}}\ S$
shows $\mu\ A\ T < \mu\ A\ S - n$
using *assms*
proof (*induction* n *arbitrary*: T)
case 0

then show ?case using *cdcl_{NOT}-measure* 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_{NOT}:
 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 - - μA])
using cdcl_{NOT}-comp-n-le[of 0 - - A] by auto

lemma rtranclp-cdcl_{NOT}-measure:

assumes
 cdcl_{NOT}** S T and
 bound-inv A S and
 cdcl_{NOT}-inv S
shows $\mu A T \leq \mu A S$
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_{NOT}-bound-inv rtranclp-imp-relpoup st step.prem)
 moreover have cdcl_{NOT}-inv T
 using c-inv rtranclp-cdcl_{NOT}-cdcl_{NOT}-inv st by blast
 ultimately have $\mu A U < \mu A T$ using cdcl_{NOT}-measure[OF - - cdcl_{NOT}] by auto
 then show ?case using IH by linarith
 qed

lemma cdcl_{NOT}-comp-bounded:

assumes
 bound-inv A S and cdcl_{NOT}-inv S and $m \geq 1 + \mu A S$
shows $\neg(\text{cdcl}_{\text{NOT}} \sim m) S T$
using assms cdcl_{NOT}-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}-restart **where**

restart-step: (cdcl_{NOT} $\sim m$) S T $\implies m \geq f n \implies \text{restart } T U$
 $\implies \text{cdcl}_{\text{NOT}}\text{-restart } (S, n) (U, \text{Suc } n) \mid$
 restart-full: full1 cdcl_{NOT} S T $\implies \text{cdcl}_{\text{NOT}}\text{-restart } (S, n) (T, \text{Suc } n)$

lemmas cdcl_{NOT}-with-restart-induct = cdcl_{NOT}-restart.induct[split-format(complete),
 OF cdcl_{NOT}-increasing-restarts-ops-axioms]

lemma *cdcl_{NOT}-restart-cdcl_{NOT}-raw-restart*:
 $cdcl_{NOT}\text{-restart } S \ T \implies cdcl_{NOT}\text{-raw-restart}^{**} (fst \ S) (fst \ T)$
proof (*induction rule*: *cdcl_{NOT}-restart.induct*)
 case (*restart-step* *m* *S* *T* *n* *U*)
 then have $cdcl_{NOT}^{**} \ S \ T$ **by** (*meson relpowp-imp-rtrancpl*)
 then have $cdcl_{NOT}\text{-raw-restart}^{**} \ S \ T$ **using** *cdcl_{NOT}-raw-restart.intros*(1)
 rtrancpl-mono[of *cdcl_{NOT}* *cdcl_{NOT}-raw-restart*] **by** *blast*
 moreover have $cdcl_{NOT}\text{-raw-restart} \ T \ U$
 using (*restart* *T* *U*) *cdcl_{NOT}-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}-raw-restart.intros*(1)
 rtrancpl-mono[of *cdcl_{NOT}* *cdcl_{NOT}-raw-restart*] **by** *auto*
qed

lemma *cdcl_{NOT}-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}-restart.induct*)
 prefer 2 **apply** (*metis rtrancpl-unfold fstI full1-def rtrancpl-cdcl_{NOT}-bound-inv*)
by (*metis cdcl_{NOT}-bound-inv cdcl_{NOT}-cdcl_{NOT}-inv cdcl_{NOT}-restart-inv fst-conv*)

lemma *cdcl_{NOT}-with-restart-cdcl_{NOT}-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}-cdcl_{NOT}-inv cdcl_{NOT}-inv-restart fst-conv*)
 apply (*metis fstI full-def full-unfold rtrancpl-cdcl_{NOT}-cdcl_{NOT}-inv*)
done

lemma *rtrancpl-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*:
assumes
 $cdcl_{NOT}\text{-restart}^{**} \ S \ T$ **and**
 $cdcl_{NOT}\text{-inv} \ (fst \ S)$
shows $cdcl_{NOT}\text{-inv} \ (fst \ T)$
using *assms* **by** *induction* (*auto intro*: *cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*)

lemma *rtrancpl-cdcl_{NOT}-with-restart-bound-inv*:
assumes
 $cdcl_{NOT}\text{-restart}^{**} \ S \ T$ **and**
 $cdcl_{NOT}\text{-inv} \ (fst \ S)$ **and**
 $bound\text{-inv} \ A \ (fst \ S)$
shows $bound\text{-inv} \ A \ (fst \ T)$
using *assms* **apply** *induction*
 apply (*simp add*: *cdcl_{NOT}-cdcl_{NOT}-inv cdcl_{NOT}-with-restart-bound-inv*)
using *cdcl_{NOT}-with-restart-bound-inv rtrancpl-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv* **by** *blast*

lemma *cdcl_{NOT}-with-restart-increasing-number:*
 $cdcl_{NOT}\text{-restart } S \ T \implies snd \ T = 1 + snd \ S$
by (induction rule: *cdcl_{NOT}-restart.induct*) *auto*
end

locale *cdcl_{NOT}-increasing-restarts =*
cdcl_{NOT}-increasing-restarts-ops restart cdcl_{NOT} f bound-inv μ cdcl_{NOT}-inv μ -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_{NOT} remove-cls_{NOT} :: '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**
 μ :: 'bound \Rightarrow 'st \Rightarrow nat **and**
cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool **and**
cdcl_{NOT}-inv :: 'st \Rightarrow bool **and**
 μ -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_{NOT}-raw-restart- μ -bound:
 $cdcl_{NOT}\text{-restart } (T, a) \ (V, b) \implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A \ T$
 $\implies \mu\text{-bound } A \ V \leq \mu\text{-bound } A \ T$
begin

lemma *rtrancp-cdcl_{NOT}-raw-restart- μ -bound:*
 $cdcl_{NOT}\text{-restart}^{**} (T, a) (V, b) \implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A \ T$
 $\implies \mu\text{-bound } A \ V \leq \mu\text{-bound } A \ T$
apply (induction rule: *rtrancp-induct2*)
apply *simp*
by (metis *cdcl_{NOT}-raw-restart- μ -bound dual-order.trans fst-conv*
rtrancp-cdcl_{NOT}-with-restart-bound-inv rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv)

lemma *cdcl_{NOT}-raw-restart-measure-bound:*
 $cdcl_{NOT}\text{-restart } (T, a) (V, b) \implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A \ T$
 $\implies \mu \ A \ V \leq \mu\text{-bound } A \ T$
apply (cases rule: *cdcl_{NOT}-restart.cases*)
apply *simp*
using *measure-bound relpowp-imp-rtrancp* **apply** *fastforce*
by (metis *full-def full-unfold measure-bound2 prod.inject*)

lemma *rtrancp-cdcl_{NOT}-raw-restart-measure-bound:*
 $cdcl_{NOT}\text{-restart}^{**} (T, a) (V, b) \implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A \ T$
 $\implies \mu \ A \ V \leq \mu\text{-bound } A \ T$
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-cdcl_{NOT}-with-restart-bound-inv rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv
rtrancp-cdcl_{NOT}-raw-restart- μ -bound)

lemma *wf-cdcl_{NOT}-restart:*
 $wf \ \{(T, S). \ cdcl_{NOT}\text{-restart } S \ T \wedge cdcl_{NOT}\text{-inv } (fst \ S)\}$ (**is** *wf ?A*)

```

proof (rule ccontr)
  assume  $\neg$  ?thesis
  then obtain  $g$  where
     $g: \bigwedge i. \text{cdcl}_{\text{NOT-restart}} (g\ i) (g\ (\text{Suc}\ i))$  and
     $\text{cdcl}_{\text{NOT-inv-g}}: \bigwedge i. \text{cdcl}_{\text{NOT-inv}} (\text{fst}\ (g\ i))$ 
    unfolding wf-iff-no-infinite-down-chain by fast

  have  $\text{snd-g}: \bigwedge i. \text{snd}\ (g\ i) = i + \text{snd}\ (g\ 0)$ 
    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  $\text{snd-g-0}: \bigwedge i. i > 0 \implies \text{snd}\ (g\ i) = i + \text{snd}\ (g\ 0)$ 
    by blast
  have  $\text{unbounded-f-g}: \text{unbounded}\ (\lambda i. f\ (\text{snd}\ (g\ i)))$ 
    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  $H: \bigwedge T\ Ta\ m. (\text{cdcl}_{\text{NOT}} \rightsquigarrow m)\ T\ Ta \implies \text{no-step}\ \text{cdcl}_{\text{NOT}}\ T \implies m = 0$ 
      apply (case-tac  $m$ ) apply simp by (meson relpowp-E2)
    have  $\exists\ T\ m. (\text{cdcl}_{\text{NOT}} \rightsquigarrow m)\ (\text{fst}\ (g\ i))\ T \wedge m \geq f\ (\text{snd}\ (g\ i))$ 
      using  $g[\text{of}\ i]$  apply (cases rule: cdclNOT-restart.cases)
      apply auto[]
      using  $g[\text{of}\ \text{Suc}\ i]$  f-ge-1 apply (cases rule: cdclNOT-restart.cases)
      apply (auto simp add: full1-def full-def dest:  $H$  dest: tranclpD)
      using  $H$  Suc-leI leD by blast
    } note  $H = \text{this}$ 
  obtain  $A$  where bound-inv  $A\ (\text{fst}\ (g\ 1))$ 
    using  $g[\text{of}\ 0]$  cdclNOT-inv-g $[\text{of}\ 0]$  apply (cases rule: cdclNOT-restart.cases)
    apply (metis One-nat-def cdclNOT-inv exists-bound fst-conv relpowp-imp-rtranclp
      rtranclp-induct)
    using  $H[\text{of}\ 1]$  unfolding full1-def by (metis One-nat-def Suc-eq-plus1 diff-is-0-eq' diff-zero
      f-ge-1 fst-conv le-add2 relpowp-E2 snd-conv)
  let  $?j = \mu\text{-bound}\ A\ (\text{fst}\ (g\ 1)) + 1$ 
  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  $\text{cdcl}_{\text{NOT-with-restart}}: j \geq i \implies \text{cdcl}_{\text{NOT-restart}}^{**} (g\ i) (g\ j)$ 
      apply (induction  $j$ )
      apply simp
      by (metis  $g$  le-Suc-eq rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
    } note  $\text{cdcl}_{\text{NOT-restart}} = \text{this}$ 
  have  $\text{cdcl}_{\text{NOT-inv}} (\text{fst}\ (g\ (\text{Suc}\ 0)))$ 
    by (simp add: cdclNOT-inv-g)
  have  $\text{cdcl}_{\text{NOT-restart}}^{**} (\text{fst}\ (g\ 1), \text{snd}\ (g\ 1)) (\text{fst}\ (g\ j), \text{snd}\ (g\ j))$ 
    using  $\langle j > 1 \rangle$  by (simp add: cdclNOT-restart)
  have  $\mu\ A\ (\text{fst}\ (g\ j)) \leq \mu\text{-bound}\ A\ (\text{fst}\ (g\ 1))$ 
    apply (rule rtranclp-cdclNOT-raw-restart-measure-bound)
    using  $\langle \text{cdcl}_{\text{NOT-restart}}^{**} (\text{fst}\ (g\ 1), \text{snd}\ (g\ 1)) (\text{fst}\ (g\ j), \text{snd}\ (g\ j)) \rangle$  apply blast
    apply (simp add: cdclNOT-inv-g)
    using  $\langle \text{bound-inv}\ A\ (\text{fst}\ (g\ 1)) \rangle$  apply simp
  done

```

```

then have  $\mu A (fst (g j)) \leq ?j$ 
  by auto
have inv: bound-inv A (fst (g j))
  using  $\langle bound-inv A (fst (g 1)) \rangle \langle cdcl_{NOT-inv} (fst (g (Suc 0))) \rangle$ 
   $\langle cdcl_{NOT-restart}^{**} (fst (g 1), snd (g 1)) (fst (g j), snd (g j)) \rangle$ 
  rtrancpl-cdclNOT-with-restart-bound-inv by auto
obtain T m where
  cdclNOT-m:  $(cdcl_{NOT} \rightsquigarrow m) (fst (g j)) T$  and
  f-m:  $f (snd (g j)) \leq m$ 
  using H[of j] by blast
have ?j < m
  using f-m j Nat.le-trans by linarith

then show False
  using  $\langle \mu A (fst (g j)) \leq \mu-bound A (fst (g 1)) \rangle$ 
  cdclNOT-comp-bounded[OF inv cdclNOT-inv-g, of ] cdclNOT-inv-g cdclNOT-m
   $\langle ?j < m \rangle$  by auto
qed

lemma cdclNOT-restart-steps-bigger-than-bound:
  assumes
    cdclNOT-restart S T and
    bound-inv A (fst S) and
    cdclNOT-inv (fst S) and
     $f (snd S) > \mu-bound A (fst S)$ 
  shows full1 cdclNOT (fst S) (fst T)
  using assms
proof (induction rule: cdclNOT-restart.induct)
  case restart-full
  then show ?case by auto
next
  case (restart-step m S T n U) note st = this(1) and f = this(2) and bound-inv = this(4) and
    cdclNOT-inv = this(5) and  $\mu = this(6)$ 
  then obtain m' where m:  $m = Suc m'$  by (cases m) auto
  have  $\mu A S - m' = 0$ 
    using f bound-inv cdclNOT-inv  $\mu m$  rtrancpl-cdclNOT-raw-restart-measure-bound by fastforce
  then have False using cdclNOT-comp-n-le[of m' S T A] restart-step unfolding m by simp
  then show ?case by fast
qed

lemma rtrancpl-cdclNOT-with-inv-inv-rtrancpl-cdclNOT:
  assumes
    inv: cdclNOT-inv S and
    binv: bound-inv A S
  shows  $(\lambda S T. cdcl_{NOT} S T \wedge cdcl_{NOT-inv} S \wedge bound-inv A S)^{**} S T \longleftrightarrow cdcl_{NOT}^{**} S T$ 
  (is  $?A^{**} S T \longleftrightarrow ?B^{**} S T$ )
  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-cdclNOT-bound-inv
    rtrancpl-cdclNOT-cdclNOT-inv)

lemma no-step-cdclNOT-restart-no-step-cdclNOT:
  assumes

```

n-s: *no-step cdcl_{NOT}-restart S* **and**
inv: *cdcl_{NOT}-inv (fst S)* **and**
binv: *bound-inv A (fst S)*
shows *no-step cdcl_{NOT} (fst S)*
proof (*rule ccontr*)
assume $\neg ?thesis$
then obtain *T* **where** *T*: *cdcl_{NOT} (fst S) T*
by *blast*
then obtain *U* **where** *U*: *full ($\lambda S T. cdcl_{NOT} S T \wedge cdcl_{NOT}\text{-inv } S \wedge bound\text{-inv } A S$) T U*
using *wf-exists-normal-form-full[OF wf-cdcl_{NOT}, of A T]* **by** *auto*
moreover have *inv-T*: *cdcl_{NOT}-inv T*
using $\langle cdcl_{NOT} (fst S) T \rangle cdcl_{NOT}\text{-inv inv}$ **by** *blast*
moreover have *b-inv-T*: *bound-inv A T*
using $\langle cdcl_{NOT} (fst S) T \rangle binv bound\text{-inv inv}$ **by** *blast*
ultimately have *full cdcl_{NOT} T U*
using *rtrancp-cdcl_{NOT}-with-inv-inv-rtrancp-cdcl_{NOT} rtrancp-cdcl_{NOT}-bound-inv*
rtrancp-cdcl_{NOT}-cdcl_{NOT}-inv **unfolding** *full-def* **by** *blast*
then have *full1 cdcl_{NOT} (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_{NOT}-merge-bj-learn-ops* =
dpll-state trail clauses prepend-trail tl-trail add-cl_s_{NOT} remove-cl_s_{NOT} +
decide-ops trail clauses prepend-trail tl-trail add-cl_s_{NOT} remove-cl_s_{NOT} +
forget-ops trail clauses prepend-trail tl-trail add-cl_s_{NOT} remove-cl_s_{NOT} forget-cond +
propagate-ops trail clauses prepend-trail tl-trail add-cl_s_{NOT} remove-cl_s_{NOT} 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-cl_s_{NOT} remove-cl_s_{NOT} :: *'v clause* \Rightarrow *'st* \Rightarrow *'st* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* \Rightarrow *'st* \Rightarrow *bool* **and**
forget-cond :: *'v clause* \Rightarrow *'st* \Rightarrow *bool* +
fixes *backjump-l-cond* :: *'v clause* \Rightarrow *'v clause* \Rightarrow *'v literal* \Rightarrow *'st* \Rightarrow *bool*
begin
inductive *backjump-l* **where**
backjump-l: *trail S = F' @ Marked K () # F*
 \Rightarrow *no-dup (trail S)*
 \Rightarrow *T* \sim *prepend-trail (Propagated L ()) (reduce-trail-to_{NOT} F (add-cl_s_{NOT} (C' + {#L#}) S))*
 \Rightarrow *C* \in *# clauses S*
 \Rightarrow *trail S* \models_{as} *CNot C*
 \Rightarrow *undefined-lit F L*
 \Rightarrow *atm-of L* \in *atms-of-msu (clauses S) \cup atm-of ' (lits-of (trail S))*
 \Rightarrow *clauses S* \models_{pm} *C' + {#L#}*
 \Rightarrow *F* \models_{as} *CNot C'*
 \Rightarrow *backjump-l-cond C C' L T*
 \Rightarrow *backjump-l S T*
inductive-cases *backjump-lE*: *backjump-l S T*
inductive *cdcl_{NOT}-merged-bj-learn* :: *'st* \Rightarrow *'st* \Rightarrow *bool* **for** *S* :: *'st* **where**

$cdcl_{NOT}\text{-merged-bj-learn-decide}_{NOT}: decide_{NOT} S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S' \mid$
 $cdcl_{NOT}\text{-merged-bj-learn-propagate}_{NOT}: propagate_{NOT} S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S' \mid$
 $cdcl_{NOT}\text{-merged-bj-learn-backjump-l}: backjump-l S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S' \mid$
 $cdcl_{NOT}\text{-merged-bj-learn-forget}_{NOT}: forget_{NOT} S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S'$

lemma $cdcl_{NOT}\text{-merged-bj-learn-no-dup-inv}$:

$cdcl_{NOT}\text{-merged-bj-learn} S T \Rightarrow no\text{-dup} (trail S) \Rightarrow no\text{-dup} (trail T)$
apply (induction rule: $cdcl_{NOT}\text{-merged-bj-learn.induct}$)
using $defined\text{-lit-map}$ **apply** $fastforce$
using $defined\text{-lit-map}$ **apply** $fastforce$
apply (force simp: $defined\text{-lit-map elim!:$ $backjump-lE$)[]
using $forget_{NOT}.simps$ **apply** $auto[1]$
done
end

locale $cdcl_{NOT}\text{-merge-bj-learn-proxy} =$

$cdcl_{NOT}\text{-merge-bj-learn-ops}$ $trail$ $clauses$ $prepend\text{-trail}$ $tl\text{-trail}$ $add\text{-cls}_{NOT}$ $remove\text{-cls}_{NOT}$
 $propagate\text{-conds}$ $forget\text{-conds}$ $\lambda C C' L' S. backjump-l\text{-cond } C C' L' S$
 $\wedge distinct\text{-mset } (C' + \{\#L'\# \}) \wedge \neg tautology (C' + \{\#L'\# \})$

for

$trail :: 'st \Rightarrow ('v, unit, unit) \text{ marked-lits}$ **and**
 $clauses :: 'st \Rightarrow 'v \text{ clauses}$ **and**
 $prepend\text{-trail} :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$ **and**
 $tl\text{-trail} :: 'st \Rightarrow 'st$ **and**
 $add\text{-cls}_{NOT} \text{ remove\text{-cls}_{NOT}} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**
 $propagate\text{-conds} :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow bool$ **and**
 $forget\text{-conds} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow bool$ **and**
 $backjump-l\text{-cond} :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow bool +$

fixes

$inv :: 'st \Rightarrow bool$

assumes

$bj\text{-can-jump}$:

$\bigwedge S C F' K F L.$

$inv S$

$\Rightarrow trail S = F' @ Marked K () \# F$

$\Rightarrow C \in \# clauses S$

$\Rightarrow trail S \models_{as} CNot C$

$\Rightarrow undefined\text{-lit } F L$

$\Rightarrow atm\text{-of } L \in atm\text{-of-msu } (clauses S) \cup atm\text{-of } ' (lits\text{-of } (F' @ Marked K () \# F))$

$\Rightarrow clauses S \models_{pm} C' + \{\#L'\# \}$

$\Rightarrow F \models_{as} CNot C'$

$\Rightarrow \neg no\text{-step } backjump-l S$ **and**

$cdcl\text{-merged-inv}: \bigwedge S T. cdcl_{NOT}\text{-merged-bj-learn} S T \Rightarrow inv S \Rightarrow inv T$

begin

abbreviation $backjump\text{-conds}$ **where**

$backjump\text{-conds} \equiv \lambda\text{- } C L \text{ - } . distinct\text{-mset } (C + \{\#L'\# \}) \wedge \neg tautology (C + \{\#L'\# \})$

sublocale $dpll\text{-with-backjumping-ops}$ $trail$ $clauses$ $prepend\text{-trail}$ $tl\text{-trail}$ $add\text{-cls}_{NOT}$ $remove\text{-cls}_{NOT}$

$propagate\text{-conds}$ inv $backjump\text{-conds}$

proof (unfold-locales, goal-cases)

case 1

{ **fix** $S S'$

assume bj : $backjump-l S S'$ **and** $no\text{-dup} (trail S)$

then obtain $F' K F L C' C$ **where**

$S': S' \sim prepend\text{-trail } (Propagated L ()) (reduce\text{-trail-to}_{NOT} F$

```

    (tl-trail(add-clsNOT (C' + {#L#}) S)))
  and
  tr-S: trail S = F' @ Marked K () # F and
  C: C ∈ # clauses S and
  tr-S-C: trail S ⊨as CNot C and
  undef-L: undefined-lit F L and
  atm-L: atm-of L ∈ atms-of-msu (clauses S) ∪ atm-of ' lits-of (trail S) and
  cls-S-C': clauses S ⊨pm C' + {#L#} and
  F-C': F ⊨as CNot C' and
  dist: distinct-mset (C' + {#L#}) and
  not-tauto: ¬ tautology (C' + {#L#})
  by (elim backjump-lE) simp

have ∃ S'. 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-eqNOT-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 cdclNOT-merge-bj-learn-proxy2 =
  cdclNOT-merge-bj-learn-proxy trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
  propagate-conds forget-conds backjump-l-cond inv
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
  forget-conds :: 'v clause ⇒ 'st ⇒ bool and
  backjump-l-cond :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ bool
begin

sublocale conflict-driven-clause-learning-ops trail clauses prepend-trail tl-trail add-clsNOT
  remove-clsNOT propagate-conds inv backjump-conds λC -. distinct-mset C ∧ ¬tautology C
  forget-conds
  by unfold-locales
end

locale cdclNOT-merge-bj-learn =

```

$cdcl_{NOT}$ -merge-bj-learn-proxy2 trail clauses prepend-trail tl-trail add-cls $_{NOT}$ remove-cls $_{NOT}$
 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-cls $_{NOT}$ remove-cls $_{NOT}$:: '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 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 $cdcl_{NOT}$:
 conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-cls $_{NOT}$ remove-cls $_{NOT}$
 propagate-conds inv backjump-conds $\lambda C. \text{distinct-mset } C \wedge \neg \text{tautology } C$ forget-conds
apply unfold-locales
apply (simp only: $cdcl_{NOT}.\text{sims}$)
using $cdcl_{NOT}$ -merged-bj-learn-forget $_{NOT}$ $cdcl$ -merged-inv learn-inv
by (auto simp add: $cdcl_{NOT}.\text{sims}$ 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 $\exists C' L. \text{learn } S (\text{add-cls}_{NOT} (C' + \{\#L\# \}) S)$
 $\wedge \text{backjump} (\text{add-cls}_{NOT} (C' + \{\#L\# \}) S) T$
 $\wedge \text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$
proof –
obtain $C F' K F L l C'$ **where**
 tr-S: trail $S = F' @ \text{Marked } K () \# F$ **and**
 T: $T \sim \text{prepend-trail } (\text{Propagated } L l) (\text{reduce-trail-to}_{NOT} F (\text{add-cls}_{NOT} (C' + \{\#L\# \}) S))$ **and**
 C-clS: $C \in \# \text{clauses } S$ **and**
 tr-S-CNot-C: trail $S \models_{as} CNot C$ **and**
 undef: undefined-lit $F L$ **and**
 atm-L: atm-of $L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$ **and**
 clss-C: clauses $S \models_{pm} C' + \{\#L\# \}$ **and**
 $F \models_{as} CNot C'$ **and**
 distinct: distinct-mset $(C' + \{\#L\# \})$ **and**
 not-tauto: $\neg \text{tautology } (C' + \{\#L\# \})$
using bt inv **by** (elim backjump-lE) simp
have atms-C': atms-of $C' \subseteq \text{atm-of ' (lits-of } F)$
proof –
obtain ll :: 'v \Rightarrow ('v literal \Rightarrow 'v) \Rightarrow 'v literal set \Rightarrow 'v literal **where**
 $\forall v f L. v \notin f ' L \vee v = f (ll v f L) \wedge ll v f L \in L$
by moura
then show ?thesis **unfolding** tr-S
by (metis (no-types) $\langle F \models_{as} CNot C' \rangle \text{atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set}$
 atms-of-def in-CNot-implies-uminus(2) mem-set-mset-iff subsetI)
qed
then have atms-of $(C' + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$
using atm-L tr-S **by** auto
moreover have learn: learn $S (\text{add-cls}_{NOT} (C' + \{\#L\# \}) S)$

```

  apply (rule learn.intros)
    apply (rule cls-C)
    using atms-C' atm-L apply (fastforce simp add: tr-S in-plus-implies-atm-of-on-atms-of-ms)[]
  apply standard
  apply (rule distinct)
  apply (rule not-tauto)
  apply simp
done
moreover have bj: backjump (add-clsNOT (C' + {#L#}) S) T
  apply (rule backjump.intros)
  using (F ⊨as CNot C') C-cls-S tr-S-CNot-C undef T distinct not-tauto n-d
  by (auto simp: tr-S state-eqNOT-def simp del: state-simpNOT)
ultimately show ?thesis by auto
qed

```

lemma *cdcl_{NOT}-merged-bj-learn-is-tranclp-cdcl_{NOT}:*
 $cdcl_{NOT}\text{-merged-bj-learn } S \ T \implies inv \ S \implies no\text{-dup } (trail \ S) \implies cdcl_{NOT}^{++} \ S \ T$

proof (induction rule: *cdcl_{NOT}-merged-bj-learn.induct*)
 case (*cdcl_{NOT}-merged-bj-learn-decide_{NOT} T*)
 then have *cdcl_{NOT} S T*
 using *bj-decide_{NOT} cdcl_{NOT}.sims* by fastforce
 then show ?case by auto
next
 case (*cdcl_{NOT}-merged-bj-learn-propagate_{NOT} T*)
 then have *cdcl_{NOT} S T*
 using *bj-propagate_{NOT} cdcl_{NOT}.sims* by fastforce
 then show ?case by auto
next
 case (*cdcl_{NOT}-merged-bj-learn-forget_{NOT} T*)
 then have *cdcl_{NOT} S T*
 using *c-forget_{NOT}* by blast
 then show ?case by auto
next
 case (*cdcl_{NOT}-merged-bj-learn-backjump-l T*) **note** *bt = this(1) and inv = this(2) and n-d = this(3)*
obtain *C' :: 'v literal multiset and L :: 'v literal where*
f3: learn S (add-cls_{NOT} (C' + {#L#}) S) ∧
backjump (add-cls_{NOT} (C' + {#L#}) S) T ∧
atms-of (C' + {#L#}) ⊆ atms-of-msu (clauses S) ∪ atm-of ' lits-of (trail S)
 using *n-d backjump-l-learn-backjump[OF bt inv]* by blast
 then have *f4: cdcl_{NOT} S (add-cls_{NOT} (C' + {#L#}) S)*
 using *n-d c-learn* by blast
 have *cdcl_{NOT} (add-cls_{NOT} (C' + {#L#}) S) T*
 using *f3 n-d bj-backjump c-dpll-bj* by blast
 then show ?case
 using *f4* by (meson tranclp.r-into-trancl tranclp.trancl-into-trancl)
qed

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}-and-inv:*
 $cdcl_{NOT}\text{-merged-bj-learn}^{**} \ S \ T \implies inv \ S \implies no\text{-dup } (trail \ S) \implies cdcl_{NOT}^{**} \ S \ T \wedge inv \ T$

proof (induction rule: *rtranclp-induct*)
 case *base*
 then show ?case by auto
next
 case (*step T U*) **note** *st = this(1) and cdcl_{NOT} = this(2) and IH = this(3)[OF this(4-)] and*

$inv = this(4)$ **and** $n-d = this(5)$
have $cdcl_{NOT}^{**} T U$
using $cdcl_{NOT}$ -merged-bj-learn-is-tranclp- $cdcl_{NOT}[OF\ cdcl_{NOT}] IH$
 $cdcl_{NOT}$.rtranclp- $cdcl_{NOT}$ -no-dup $inv\ n-d$ **by** *auto*
then have $cdcl_{NOT}^{**} S U$ **using** IH **by** *fastforce*
moreover have $inv\ U$ **using** $n-d\ IH\ \langle cdcl_{NOT}^{**} T\ U \rangle\ cdcl_{NOT}$.rtranclp- $cdcl_{NOT}$ - inv **by** *blast*
ultimately show *?case* **using** *st* **by** *fast*
qed

lemma $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$:
 $cdcl_{NOT}$ -merged-bj-learn $^{**} S\ T \implies inv\ S \implies no-dup\ (trail\ S) \implies cdcl_{NOT}^{**} S\ T$
using $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$ -and- inv **by** *blast*

lemma $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn- inv :
 $cdcl_{NOT}$ -merged-bj-learn $^{**} S\ T \implies inv\ S \implies no-dup\ (trail\ S) \implies inv\ T$
using $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$ -and- inv **by** *blast*

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

definition μ_{CDCL}' -merged $:: 'v\ literal\ multiset\ set \Rightarrow 'st \Rightarrow nat$ **where**
 μ_{CDCL}' -merged $A\ T \equiv$
 $((2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A)) - \mu_C' A\ T) * 2 + card\ (set-mset\ (clauses\ T))$

lemma $cdcl_{NOT}$ -decreasing-measure':
assumes
 $cdcl_{NOT}$ -merged-bj-learn $S\ T$ **and**
 $inv: inv\ S$ **and**
 $atm-clss: atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A$ **and**
 $atm-trail: atm-of\ ' lits-of\ (trail\ S) \subseteq atms-of-ms\ A$ **and**
 $n-d: no-dup\ (trail\ S)$ **and**
 $fin-A: finite\ A$
shows μ_{CDCL}' -merged $A\ T < \mu_{CDCL}'$ -merged $A\ S$
using *assms(1)*

proof *induction*

case ($cdcl_{NOT}$ -merged-bj-learn-decide $_{NOT}$ T)
have $clauses\ S = clauses\ T$
using $cdcl_{NOT}$ -merged-bj-learn-decide $_{NOT}$.*hyps* **by** *auto*
moreover have
 $(2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $- \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ T)$
 $< (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $- \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ S)$
apply (*rule dpll-bj-trail-mes-decreasing-prop*)
using $cdcl_{NOT}$ -merged-bj-learn-decide $_{NOT}$ $fin-A\ atm-clss\ atm-trail\ n-d\ inv$
by (*simp-all add: bj-decide $_{NOT}$ cdcl $_{NOT}$ -merged-bj-learn-decide $_{NOT}$.hyps*)
ultimately show *?case*
unfolding μ_{CDCL}' -merged-def μ_C' -def **by** *simp*

next

case ($cdcl_{NOT}$ -merged-bj-learn-propagate $_{NOT}$ T)
have $clauses\ S = clauses\ T$
using $cdcl_{NOT}$ -merged-bj-learn-propagate $_{NOT}$.*hyps*
by (*simp add: bj-propagate $_{NOT}$ inv dpll-bj-clauses*)
moreover have
 $(2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$

$\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $< (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$
apply (rule *dpll-bj-trail-mes-decreasing-prop*)
using *inv n-d atm-clss atm-trail fin-A* **by** (*simp-all add: bj-propagate_{NOT}*
cdcl_{NOT}-merged-bj-learn-propagate_{NOT}.hyps)
ultimately show ?*case*
unfolding $\mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def}$ **by** *simp*
next
case (*cdcl_{NOT}-merged-bj-learn-forget_{NOT} T*)
have $\text{card} (\text{set-mset} (\text{clauses } T)) < \text{card} (\text{set-mset} (\text{clauses } S))$
using $\langle \text{forget}_{NOT} S T \rangle$ **by** (*metis card-Diff1-less*
cdcl_{NOT}-merged-bj-learn-forget_{NOT}.hyps clauses-remove-cls_{NOT} finite-set-mset forgetE
mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
moreover
have $\text{trail } S = \text{trail } T$
using $\langle \text{forget}_{NOT} S T \rangle$ **by** (*auto elim: forgetE*)
then have
 $(2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $= (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$
by *auto*
ultimately show ?*case*
unfolding $\mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def}$ **by** *simp*
next
case (*cdcl_{NOT}-merged-bj-learn-backjump-l T*) **note** *bj-l = this(1)*
obtain $C' L$ **where**
learn: learn S (add-cls_{NOT} (C' + {#L#}) S) and
bj: backjump (add-cls_{NOT} (C' + {#L#}) S) T and
atms-C: atms-of (C' + {#L#}) \subseteq atms-of-msu (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 $\text{card-}T\text{-}S: \text{card} (\text{set-mset} (\text{clauses } T)) \leq 1 + \text{card} (\text{set-mset} (\text{clauses } S))$
using *bj-l inv* **by** (*force elim!: backjump-lE simp: card-insert-if*)
have
 $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T))$
 $< ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A))$
 $(\text{trail-weight} (\text{add-cls}_{NOT} (C' + \{ \#L\# \}) S)))$
apply (rule *dpll-bj-trail-mes-decreasing-prop*)
using *bj bj-backjump* **apply** *blast*
using *cdcl_{NOT}.c-learn cdcl_{NOT}.cdcl_{NOT}-inv inv learn* **apply** *blast*
using *atms-C atm-clss atm-trail n-d clauses-add-cls_{NOT}* **apply** *simp* **apply** *fast*
using *atm-trail n-d* **apply** *simp*
apply (*simp add: n-d*)
using *fin-A* **apply** *simp*
done
then have $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T))$
 $< ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S))$
using *n-d* **by** *auto*
then show ?*case*
using *card-T-S* **unfolding** $\mu_{CDCL}'\text{-merged-def } \mu_C'\text{-def}$ **by** *linarith*

qed

lemma *wf-cdcl_{NOT}-merged-bj-learn*:

assumes

fin-A: finite *A*

shows *wf* $\{(T, S).$

$(inv\ S \wedge atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}$

apply (rule *wfP-if-measure*[of - - μ_{CDCL}' -merged *A*])

using *cdcl_{NOT}-decreasing-measure'* *fin-A* **by** *simp*

lemma *tranclp-cdcl_{NOT}-cdcl_{NOT}-tranclp*:

assumes

cdcl_{NOT}-merged-bj-learn⁺⁺ *S T* **and**

inv: *inv S* **and**

atm-clss: *atms-of-msu* (*clauses S*) \subseteq *atms-of-ms A* **and**

atm-trail: *atm-of* ' *lits-of* (*trail S*) \subseteq *atms-of-ms A* **and**

n-d: *no-dup* (*trail S*) **and**

fin-A[*simp*]: finite *A*

shows $(T, S) \in \{(T, S).$

$(inv\ S \wedge atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}^+ \text{ (is - } \in ?P^+)$

using *assms*(1)

proof (*induction rule*: *tranclp-induct*)

case *base*

then show ?*case* **using** *n-d atm-clss atm-trail inv* **by** *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-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}*)

using *st cdcl_{NOT} inv n-d atm-clss atm-trail inv* **by** *auto*

have *inv T*

apply (rule *rtranclp-cdcl_{NOT}-merged-bj-learn-inv*)

using *inv st cdcl_{NOT} n-d atm-clss atm-trail inv* **by** *auto*

moreover have *atms-of-msu* (*clauses T*) \subseteq *atms-of-ms A*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound*[*OF* $\langle cdcl_{NOT}^{**}\ S\ T \rangle$ *inv n-d atm-clss atm-trail*]
by *fast*

moreover have *atm-of* ' (*lits-of* (*trail T*)) \subseteq *atms-of-ms A*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound*[*OF* $\langle cdcl_{NOT}^{**}\ S\ T \rangle$ *inv n-d atm-clss atm-trail*]
by *fast*

moreover have *no-dup* (*trail T*)

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-no-dup*[*OF* $\langle cdcl_{NOT}^{**}\ S\ T \rangle$ *inv n-d*] **by** *fast*

ultimately have (*U, T*) $\in ?P$

using *cdcl_{NOT}* **by** *auto*

then show ?*case* **using** *IH* **by** (*simp add*: *trancl-into-trancl2*)

qed

lemma *wf-tranclp-cdcl_{NOT}-merged-bj-learn*:

assumes finite *A*

shows *wf* $\{(T, S).$

$(inv\ S \wedge atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}^{++}\ S\ T\}$

```

apply (rule wf-subset)
apply (rule wf-trancl[OF wf-cdclNOT-merged-bj-learn])
using assms apply simp
using tranclp-cdclNOT-cdclNOT-tranclp[OF - - - -  $\langle \text{finite } A \rangle$ ] by auto

```

```

lemma backjump-no-step-backjump-l:
  backjump  $S$   $T \implies \text{inv } S \implies \neg \text{no-step backjump-l } S$ 
apply (elim backjumpE)
apply (rule bj-can-jump)
  apply auto[7]
by blast

```

```

lemma cdclNOT-merged-bj-learn-final-state:
  fixes  $A :: \text{'v literal multiset set}$  and  $S$   $T :: \text{'st}$ 
assumes
    n-s: no-step cdclNOT-merged-bj-learn  $S$  and
    atms-S: atms-of-msu (clauses  $S$ )  $\subseteq$  atms-of-ms  $A$  and
    atms-trail: atm-of ' lits-of (trail  $S$ )  $\subseteq$  atms-of-ms  $A$  and
    n-d: no-dup (trail  $S$ ) and
    finite  $A$  and
    inv: inv  $S$  and
    decomp: all-decomposition-implies-m (clauses  $S$ ) (get-all-marked-decomposition (trail  $S$ ))
shows unsatisfiable (set-mset (clauses  $S$ ))
   $\vee$  (trail  $S \models_{\text{asm}}$  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  $\models_{\text{as}}$  ?N
  | (sat') satisfiable ?N and  $\neg ?M \models_{\text{as}}$  ?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  $I :: \text{'v literal set}$  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-ms ?N
    using sat unfolding satisfiable-def-min by auto
    let ?I =  $I \cup \{P \mid P. P \in \text{lits-of } ?M \wedge \text{atm-of } P \notin \text{atm-of ' } I\}$ 
    let ?O =  $\{\{\# \text{lit-of } L \# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M \wedge \text{atm-of (lit-of } L) \notin \text{atms-of-ms } ?N\}$ 
    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 ( $?N \cup (\lambda a. \{\# \text{lit-of } a \# \})$  ' set ?M)
      using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
      by fastforce
    have  $\{P \mid P. P \in \text{lits-of } ?M \wedge \text{atm-of } P \notin \text{atm-of ' } I\} \models_s ?O$ 
      using  $\langle I \models_s ?N \rangle$  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  $\langle I \models_s ?N \rangle$  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-ms ?N ⊆ atm-of 'lits-of ?M
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain l :: 'v where
    l-N: l ∈ atms-of-ms ?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-ms-mono atms-of-ms-CNot-atms-of atms-of-ms-CNot-atms-of-ms 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-ms ?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-ms-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-ms ?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 ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ cons-I' atms-N-M
    by (meson (trail S ⊨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 literal and d :: unit and
  F F' :: ('v, unit, unit) marked-lit list where
  M-K: ?M = F' @ Marked K () # F and
  nm: ∀ f ∈ set F'. ¬ is-marked f
  unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let ?K = Marked K () :: ('v, unit, unit) marked-lit
have ?K ∈ set ?M
  unfolding M-K by auto

```

```

let ?C = image-mset lit-of {#L ∈ #mset ?M. is-marked L ∧ L ≠ ?K#} :: 'v literal multiset
let ?C' = set-mset (image-mset (λL::'v literal. {#L#}) (?C + {#lit-of ?K#}))
have ?N ∪ {{#lit-of L#} | L. is-marked L ∧ L ∈ set ?M} ⊨ps (λa. {#lit-of a#}) ' set ?M
  using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' = {{#lit-of L#} | L. is-marked L ∧ L ∈ set ?M}
  unfolding M-K apply standard
  apply force
  using IntI by auto
ultimately have N-C-M: ?N ∪ ?C' ⊨ps (λa. {#lit-of a#}) ' set ?M
  by auto
have N-M-False: ?N ∪ (λL. {#lit-of L#}) ' (set ?M) ⊨ps {{#}}
  using M ⟨?M ⊨as CNot C⟩ ⟨C ∈ ?N⟩ unfolding true-clss-clss-def true-annot-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 ⟨no-dup ?M⟩ unfolding M-K by (simp add: defined-lit-map)
moreover
  have ?N ∪ ?C' ⊨ps {{#}}
  proof -
    have A: ?N ∪ ?C' ∪ (λa. {#lit-of a#}) ' set ?M =
      ?N ∪ (λa. {#lit-of a#}) ' 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 ⊨p 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-ms (?N ∪ {image-mset uminus ?C + {#-K#}})) and
      cons: consistent-interp I and
      I ⊨s ?N
    have (K ∈ I ∧ -K ∉ I) ∨ (-K ∈ I ∧ K ∉ I)
      using cons tot unfolding consistent-interp-def by (cases K) auto
    have tot': total-over-set I
      (atm-of ' lit-of ' (set ?M ∩ {L. is-marked L ∧ L ≠ 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 ∉ I and
        a1: x ∈ 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-ms-single-image-atm-of-lit-of)
  then show  $I \models \text{image-mset } \text{uminus } ?C + \{\# - K \#\}$ 
    unfolding true-clss-def true-cl-def Bex-mset-def
    using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
    by (auto dest!: H)
  qed
  moreover have  $F \models_{as} CNot (\text{image-mset } \text{uminus } ?C)$ 
    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  $\{\# L : \# \text{ mset } ?M. \text{is-marked } L \wedge L \neq \text{Marked } K ()\#\}$ )
       $\langle C \in ?N \rangle n-s \langle ?M \models_{as} CNot C \rangle$  bj-backjump inv unfolding M-K
      by (auto simp: cdclNOT-merged-bj-learn.simps)
    then show ?thesis by fast
  qed auto
qed

```

lemma *full-cdcl_{NOT}-merged-bj-learn-final-state:*

fixes $A :: 'v \text{ literal multiset set}$ **and** $S \ T :: 'st$

assumes

full: *full cdcl_{NOT}-merged-bj-learn S T and*

atms-S: *atms-of-msu (clauses S) \subseteq atms-of-ms A and*

atms-trail: *atm-of ' lits-of (trail S) \subseteq atms-of-ms A and*

n-d: *no-dup (trail S) and*

finite A and

inv: *inv S and*

decomp: *all-decomposition-implies-m (clauses 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_{NOT}-merged-bj-learn** S T and n-s:* *no-step cdcl_{NOT}-merged-bj-learn T*

using *full* **unfolding** *full-def* **by** *blast+*

then have *st:* *cdcl_{NOT}** S T*

using *inv rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}-and-inv n-d* **by** *auto*

have *atms-of-msu (clauses T) \subseteq atms-of-ms A and atm-of ' lits-of (trail T) \subseteq atms-of-ms A*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound[OF st inv n-d atms-S atms-trail]* **by** *blast+*

moreover have *no-dup (trail T)*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-no-dup inv n-d st* **by** *blast*

moreover have *inv T*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-inv inv st* **by** *blast*

moreover have *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-all-decomposition-implies inv st decomp n-d* **by** *blast*

ultimately show *?thesis*

using *cdcl_{NOT}-merged-bj-learn-final-state[of T A] \langle finite A \rangle n-s* **by** *fast*

qed

end

14.8.1 Instantiations

locale *cdcl_{NOT}-with-backtrack-and-restarts =*

conflict-driven-clause-learning-learning-before-backjump-only-distinct-learned trail clauses

prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} propagate-conds inv backjump-conds

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-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 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:  $\bigwedge S\ T. inv\ S \Rightarrow T \sim reduce\_trail\_to_{NOT} ([::'a\ list)\ S \Rightarrow inv\ T$ 
begin

lemma bound-inv-inv:
assumes
  inv S and
  n-d: no-dup (trail S) and
  atms-clss-S-A: atms-of-msu (clauses S)  $\subseteq$  atms-of-ms A and
  atms-trail-S-A: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-ms A and
  finite A and
  cdclNOT: cdclNOT S T
shows
  atms-of-msu (clauses T)  $\subseteq$  atms-of-ms A and
  atm-of ' lits-of (trail T)  $\subseteq$  atms-of-ms A and
  finite A
proof –
  have cdclNOT S T
    using  $\langle inv\ S \rangle$  cdclNOT by linarith
  then have atms-of-msu (clauses T)  $\subseteq$  atms-of-msu (clauses S)  $\cup$  atm-of ' lits-of (trail S)
    using  $\langle inv\ S \rangle$ 
    by (meson conflict-driven-clause-learning-ops.cdclNOT-atms-of-ms-clauses-decreasing
      conflict-driven-clause-learning-ops-axioms n-d)
  then show atms-of-msu (clauses T)  $\subseteq$  atms-of-ms A
    using atms-clss-S-A atms-trail-S-A by blast
next
  show atm-of ' lits-of (trail T)  $\subseteq$  atms-of-ms A
    by (meson  $\langle inv\ S \rangle$  atms-clss-S-A atms-trail-S-A cdclNOT cdclNOT-atms-in-trail-in-set n-d)
next
  show finite A
    using  $\langle finite\ A \rangle$  by simp
qed

sublocale cdclNOT-increasing-restarts-ops  $\lambda S\ T. T \sim reduce\_trail\_to_{NOT} ([::'a\ list)\ S\ cdcl_{NOT}\ f$ 
 $\lambda A\ S. atms\_of\_msu\ (clauses\ S) \subseteq atms\_of\_ms\ A \wedge atm\_of\ ' \ lits\_of\ (trail\ S) \subseteq atms\_of\_ms\ A \wedge$ 
finite A
 $\mu_{CDCL}'\ \lambda S. inv\ S \wedge no\_dup\ (trail\ S)$ 
 $\mu_{CDCL}'$ -bound
apply unfold-locales
  apply (simp add: unbounded)
  using f-ge-1 apply force
  using bound-inv-inv apply meson
  apply (rule cdclNOT-decreasing-measure'; simp)

```



```

    apply (rule rtrancpl-cdclNOT-μCDCL'-bound; simp)
    apply (rule rtrancpl-μCDCL'-bound-decreasing; simp)
    apply auto[]
    apply auto[]
    using cdclNOT-inv cdclNOT-no-dup apply blast
    using inv-restart apply auto[]
done

```

abbreviation *cdcl_{NOT}-l* **where**

```

cdclNOT-l ≡
  conflict-driven-clause-learning-ops.cdclNOT trail clauses prepend-trail tl-trail add-clsNOT
  remove-clsNOT propagate-conds (λ- - - S T. backjump S T)
  (λC S. distinct-mset C ∧ ¬ tautology C ∧ learn-restrictions C S
    ∧ (∃ F K F' C' L. trail S = F' @ Marked K () # F ∧ C = C' + {#L#}
      ∧ F ⊨as CNot C' ∧ C' + {#L#} ∉ # clauses S))
  (λC S. ¬ (∃ F' F K L. trail S = F' @ Marked K () # F ∧ F ⊨as CNot (C - {#L#})))
  ∧ forget-restrictions C S)

```

lemma *cdcl_{NOT}-with-restart-μ_{CDC}L'-le-μ_{CDC}L'-bound*:

```

assumes
  cdclNOT: cdclNOT-restart (T, a) (V, b) and
  cdclNOT-inv:
    inv T
    no-dup (trail T) and
  bound-inv:
    atms-of-msu (clauses T) ⊆ atms-of-ms A
    atm-of ' lits-of (trail T) ⊆ atms-of-ms A
    finite A

```

shows μ_{CDC}L' A V ≤ μ_{CDC}L'-bound A T

using cdcl_{NOT}-inv bound-inv

proof (induction rule: *cdcl_{NOT}-with-restart-induct*[OF *cdcl_{NOT}*])

case (1 m S T n U) **note** U = *this*(3)

show ?case

```

  apply (rule rtrancpl-cdclNOT-μCDCL'-bound-reduce-trail-toNOT[of S T])
    using <(cdclNOT ~ m) S T> apply (fastforce dest!: relpowp-imp-rtrancpl)
    using 1 by auto

```

next

case (2 S T n) **note** full = *this*(2)

show ?case

```

  apply (rule rtrancpl-cdclNOT-μCDCL'-bound)
  using full 2 unfolding full1-def by force+

```

qed

lemma *cdcl_{NOT}-with-restart-μ_{CDC}L'-bound-le-μ_{CDC}L'-bound*:

```

assumes
  cdclNOT: cdclNOT-restart (T, a) (V, b) and
  cdclNOT-inv:
    inv T
    no-dup (trail T) and
  bound-inv:
    atms-of-msu (clauses T) ⊆ atms-of-ms A
    atm-of ' lits-of (trail T) ⊆ atms-of-ms A
    finite A

```

shows μ_{CDC}L'-bound A V ≤ μ_{CDC}L'-bound A T

using cdcl_{NOT}-inv bound-inv

proof (*induction rule: cdcl_{NOT}-with-restart-induct*[*OF cdcl_{NOT}*])
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- μ_{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- μ_{CDCL}' -bound-decreasing*)
using *full 2* **unfolding** *full1-def* **by** *force+*
qed

sublocale *cdcl_{NOT}-increasing-restarts - - - - - f*
 $\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} (\ [] :: 'a \ \text{list}) \ S$
 $\lambda A \ S. \ \text{atms-of-msu} (\ \text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of } ' \ \text{lits-of } (\ \text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}' \ cdcl_{NOT}$
 $\lambda S. \ \text{inv } S \wedge \text{no-dup } (\ \text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply *unfold-locale*
using *cdcl_{NOT}-with-restart- μ_{CDCL}' -le- μ_{CDCL}' -bound* **apply** *simp*
using *cdcl_{NOT}-with-restart- μ_{CDCL}' -bound-le- μ_{CDCL}' -bound* **apply** *simp*
done

lemma *cdcl_{NOT}-restart-all-decomposition-implies:*
assumes *cdcl_{NOT}-restart S T and*
inv (fst S) and
no-dup (trail (fst S))
all-decomposition-implies-m (clauses (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_{NOT}-all-decomposition-implies* **by** (*auto dest!: trancp-into-rtrancp*
simp: full1-def)

lemma *rtrancp-cdcl_{NOT}-restart-all-decomposition-implies:*
assumes *cdcl_{NOT}-restart** S T and*
inv: inv (fst S) and
n-d: no-dup (trail (fst S)) and
decomp:
all-decomposition-implies-m (clauses (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: rtrancp-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 *rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*[*OF st*] *inv n-d* **by** *blast*
moreover have *no-dup (trail (fst T))*
using *rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*[*OF st*] *inv n-d* **by** *blast*

ultimately show ?case
using *cdcl_{NOT}-restart-all-decomposition-implies r IH n-d* **by** *fast*
qed

lemma *cdcl_{NOT}-restart-sat-ext-iff*:

assumes

st: *cdcl_{NOT}-restart S T* **and**
n-d: *no-dup (trail (fst S))* **and**
inv: *inv (fst S)*

shows $I \models_{\text{sextm}} \text{clauses } (fst S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}(fst T)$

using *assms*

proof (*induction*)

case (*restart-step m S T n U*)

then show ?case

using *rtrancpl-cdcl_{NOT}-bj-sat-ext-iff n-d* **by** (*fastforce dest!: relpowp-imp-rtrancpl*)

next

case *restart-full*

then show ?case **using** *rtrancpl-cdcl_{NOT}-bj-sat-ext-iff* **unfolding** *full1-def*

by (*fastforce dest!: trancpl-into-rtrancpl*)

qed

lemma *rtrancpl-cdcl_{NOT}-restart-sat-ext-iff*:

assumes

st: *cdcl_{NOT}-restart** S T* **and**
n-d: *no-dup (trail (fst S))* **and**
inv: *inv (fst S)*

shows $I \models_{\text{sextm}} \text{clauses } (fst S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}(fst T)$

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_{NOT}-with-restart-cdcl_{NOT}-inv[OF st]* *inv n-d* **by** *blast+*

moreover have *no-dup (trail (fst T))*

using *rtrancpl-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv* *rtrancpl-cdcl_{NOT}-no-dup st inv n-d* **by** *blast*

ultimately show ?case

using *cdcl_{NOT}-restart-sat-ext-iff[OF r]* *IH* **by** *blast*

qed

theorem *full-cdcl_{NOT}-restart-backjump-final-state*:

fixes *A :: 'v literal multiset set* **and** *S T :: 'st*

assumes

full: *full cdcl_{NOT}-restart (S, n) (T, m)* **and**
atms-S: *atms-of-msu (clauses S) \subseteq atms-of-ms A* **and**
atms-trail: *atm-of ' lits-of (trail S) \subseteq atms-of-ms A* **and**
n-d: *no-dup (trail S)* **and**
fin-A[simp]: *finite A* **and**
inv: *inv S* **and**

decomp: *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

shows *unsatisfiable (set-mset (clauses S))*

\vee (*lits-of (trail T) \models_{sextm} clauses S \wedge satisfiable (set-mset (clauses S))*)

proof –

have *st*: *cdcl_{NOT}-restart** (S, n) (T, m)* **and**

```

  n-s: no-step cdclNOT-restart (T, m)
  using full unfolding full-def by fast+
have binv-T: atms-of-msu (clauses T)  $\subseteq$  atms-of-ms A atm-of ' lits-of (trail T)  $\subseteq$  atms-of-ms A
  using rtrancp-cdclNOT-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 rtrancp-cdclNOT-with-restart-cdclNOT-inv[OF st] inv n-d by auto
moreover have all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))
  using rtrancp-cdclNOT-restart-all-decomposition-implies[OF st] inv n-d
  decomp by auto
ultimately have T: unsatisfiable (set-mset (clauses T))
   $\vee$  (trail T  $\models_{asm}$  clauses T  $\wedge$  satisfiable (set-mset (clauses T)))
  using no-step-cdclNOT-restart-no-step-cdclNOT[of (T, m) A] n-s
  cdclNOT-final-state[of T A] unfolding cdclNOT-NOT-all-inv-def by auto
have eq-sat-S-T:  $\bigwedge I. I \models_{sextm} \text{clauses } S \longleftrightarrow I \models_{sextm} \text{clauses } T$ 
  using rtrancp-cdclNOT-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  $\models_{asm}$  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-clss-ext
    unfolding satisfiable-def by blast
  then show ?thesis by fast
next
  case sat
  then have lits-of (trail T)  $\models_{sextm}$  clauses S
    using rtrancp-cdclNOT-restart-sat-ext-iff[OF st] inv n-d atms-S
    atms-trail by (auto simp: true-clss-imp-true-clss-ext true-annots-true-clss)
  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 cdclNOT-with-backtrack-and-restarts locale

```

```

locale most-general-cdclNOT =
  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  $\lambda$  - - - . True
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
begin

```

lemma *backjump-bj-can-jump*:

assumes

tr-S: $\text{trail } S = F' @ \text{Marked } K () \# F$ **and**

C: $C \in \# \text{ clauses } S$ **and**

tr-S-C: $\text{trail } S \models_{\text{as}} C \text{Not } C$ **and**

undef: *undefined-lit* *F L* **and**

atm-L: $\text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (F' @ \text{Marked } K () \# F))$ **and**

cls-S-C': $\text{clauses } S \models_{\text{pm}} C' + \{\#L\#\}$ **and**

F-C': $F \models_{\text{as}} C \text{Not } C'$

shows $\neg \text{no-step backjump } S$

using *backjump.intros*[*OF tr-S - C tr-S-C undef - cls-S-C' F-C'*,

of prepend-trail (Propagated L -) (reduce-trail-to_{NOT} F S)] atm-L unfolding tr-S

by (*auto simp: state-eq_{NOT}-def simp del: state-simp_{NOT}*)

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 *cdcl_{NOT}-merge-bj-learn-with-backtrack-restarts* =

cdcl_{NOT}-merge-bj-learn *trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT}*

propagate-conds inv forget-conds

$\lambda C C' L' S. \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \text{backjump-l-cond } C C' L' S$

for

trail :: $'st \Rightarrow ('v::\text{linorder}, \text{unit}, \text{unit}) \text{ marked-lits}$ **and**

clauses :: $'st \Rightarrow 'v::\text{linorder} \text{ clauses}$ **and**

prepend-trail :: $('v, \text{unit}, \text{unit}) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$ **and**

tl-trail :: $'st \Rightarrow 'st$ **and**

add-cls_{NOT} remove-cls_{NOT} :: $'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**

propagate-conds :: $('v, \text{unit}, \text{unit}) \text{ marked-lit} \Rightarrow 'st \Rightarrow \text{bool}$ **and**

inv :: $'st \Rightarrow \text{bool}$ **and**

forget-conds :: $'v \text{ clause} \Rightarrow 'st \Rightarrow \text{bool}$ **and**

backjump-l-cond :: $'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow \text{bool}$

+

fixes *f* :: $\text{nat} \Rightarrow \text{nat}$

assumes

unbounded: *unbounded f* **and** *f-ge-1*: $\bigwedge n. n \geq 1 \Rightarrow f n \geq 1$ **and**

inv-restart: $\bigwedge S T. \text{inv } S \Rightarrow T \sim \text{reduce-trail-to}_{\text{NOT}} [] S \Rightarrow \text{inv } T$

begin

interpretation *cdcl_{NOT}*:

conflict-driven-clause-learning-ops *trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT}*

propagate-conds inv backjump-conds ($\lambda C -. \text{distinct-mset } C \wedge \neg \text{tautology } C$) *forget-conds*

by *unfold-locales*

interpretation *cdcl_{NOT}*:

conflict-driven-clause-learning *trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT}*

propagate-conds inv backjump-conds ($\lambda C -. \text{distinct-mset } C \wedge \neg \text{tautology } C$) *forget-conds*

apply *unfold-locales*

using *cdcl_{NOT}-merged-bj-learn-forget_{NOT} cdcl-merged-inv learn-inv*

by (*auto simp add: cdcl_{NOT}.simps dpll-bj-inv*)

definition *not-simplified-cl* $A = \{\#C \in \# A. \text{tautology } C \vee \neg \text{distinct-mset } C\# \}$

lemma *build-all-simple-clss-or-not-simplified-cl*:

assumes *atms-of-msu* (*clauses* S) \subseteq *atms-of-ms* A **and**

$x \in \# \text{clauses } S$ **and** *finite* A

shows $x \in \text{build-all-simple-clss } (\text{atms-of-ms } A) \vee x \in \# \text{not-simplified-cl } (\text{clauses } S)$

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 $x \in \text{build-all-simple-clss } (\text{atms-of-ms } A)$

by (*meson* *assms* *atms-of-atms-of-ms-mono* *atms-of-ms-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 $x \in \# \text{not-simplified-cl } (\text{clauses } S)$

using $\langle x \in \# \text{clauses } S \rangle$ **unfolding** *not-simplified-cl-def* **by** *auto*

then show ?thesis **by** *blast*

qed

qed

lemma *cdcl_{NOT}-merged-bj-learn-clauses-bound*:

assumes

cdcl_{NOT}-merged-bj-learn S T **and**

inv: *inv* S **and**

atms-clss: *atms-of-msu* (*clauses* S) \subseteq *atms-of-ms* A **and**

atms-trail: *atm-of* (*lits-of* (*trail* S)) \subseteq *atms-of-ms* A **and**

n-d: *no-dup* (*trail* S) **and**

fin-A[*simpl*]: *finite* A

shows *set-mset* (*clauses* T) \subseteq *set-mset* (*not-simplified-cl* (*clauses* S))

$\cup \text{build-all-simple-clss } (\text{atms-of-ms } A)$

using *assms*

proof (*induction rule*: *cdcl_{NOT}-merged-bj-learn.induct*)

case *cdcl_{NOT}-merged-bj-learn-decide_{NOT}*

then show ?case **using** *dpll-bj-clauses* **by** (*force* *dest*!: *build-all-simple-clss-or-not-simplified-cl*)

next

case *cdcl_{NOT}-merged-bj-learn-propagate_{NOT}*

then show ?case **using** *dpll-bj-clauses* **by** (*force* *dest*!: *build-all-simple-clss-or-not-simplified-cl*)

next

case *cdcl_{NOT}-merged-bj-learn-forget_{NOT}*

then show ?case **using** *clauses-remove-cl_{NOT}* **unfolding** *state-eq_{NOT}-def*

by (*force* *elim*!: *forgetE* *dest*: *build-all-simple-clss-or-not-simplified-cl*)

next

case (*cdcl_{NOT}-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_{NOT}*** S T

apply (*rule* *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}*)

using $\langle \text{backjump-l } S \ T \rangle$ *inv* *cdcl_{NOT}-merged-bj-learn.simps* *n-d* **by** *blast*+

have $\text{atm-of } \langle \text{lits-of } (\text{trail } T) \rangle \subseteq \text{atms-of-ms } A$
using $\text{cdcl}_{NOT}.\text{rtrancpl-cdcl}_{NOT}\text{-trail-clauses-bound}[OF \langle \text{cdcl}_{NOT}^{**} S T \rangle \text{ inv atms-trail atms-clss } n\text{-d}]$ **by** *auto*
have $\text{atms-of-msu } (\text{clauses } T) \subseteq \text{atms-of-ms } A$
using $\text{cdcl}_{NOT}.\text{rtrancpl-cdcl}_{NOT}\text{-trail-clauses-bound}[OF \langle \text{cdcl}_{NOT}^{**} S T \rangle \text{ inv } n\text{-d atms-clss atms-trail}]$ **by** *fast*
moreover have $\text{no-dup } (\text{trail } T)$
using $\text{cdcl}_{NOT}.\text{rtrancpl-cdcl}_{NOT}\text{-no-dup}[OF \langle \text{cdcl}_{NOT}^{**} S T \rangle \text{ inv } n\text{-d}]$ **by** *fast*

obtain $F' K F L l C' C$ **where**
tr-S: $\text{trail } S = F' @ \text{Marked } K () \# F$ **and**
T: $T \sim \text{prepend-trail } (\text{Propagated } L l) (\text{reduce-trail-to}_{NOT} F (\text{add-cl}_{NOT} (C' + \{\#L\# \}) S))$ **and**
 $C \in \# \text{ clauses } S$ **and**
 $\text{trail } S \models_{as} CNot C$ **and**
undef: $\text{undefined-lit } F L$ **and**
 $\text{atm-of } L = \text{atm-of } K \vee \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S)$
 $\vee \text{atm-of } L \in \text{atm-of } \langle \text{lits-of } F' \cup \text{lits-of } F \rangle$ **and**
 $\text{clauses } S \models_{pm} C' + \{\#L\# \}$ **and**
 $F \models_{as} CNot C'$ **and**
dist: $\text{distinct-mset } (C' + \{\#L\# \})$ **and**
tauto: $\neg \text{tautology } (C' + \{\#L\# \})$ **and**
backjump-l-cond $C C' L T$
using $\langle \text{backjump-l } S T \rangle$ **apply** (*induction rule: backjump-l.induct*) **by** *auto*

have $\text{atms-of } C' \subseteq \text{atm-of } \langle \text{lits-of } F \rangle$
using $\langle F \models_{as} CNot C' \rangle$ **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 $\text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-ms } A$
using $T \langle \text{atm-of } \langle \text{lits-of } (\text{trail } T) \rangle \subseteq \text{atms-of-ms } A \rangle$ *tr-S undef n-d* **by** *auto*
then have $\text{build-all-simple-clss } (\text{atms-of } (C' + \{\#L\# \})) \subseteq \text{build-all-simple-clss } (\text{atms-of-ms } A)$
apply – **by** (*rule build-all-simple-clss-mono*) (*simp-all*)
then have $C' + \{\#L\# \} \in \text{build-all-simple-clss } (\text{atms-of-ms } A)$
using $\text{distinct-mset-not-tautology-implies-in-build-all-simple-clss}[OF \text{ dist tauto}]$
by *auto*
then show *?case*
using $T \text{ inv atms-clss undef tr-S } n\text{-d}$
by (*force dest!: build-all-simple-clss-or-not-simplified-cls*)

qed

lemma $\text{cdcl}_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$:
assumes $\text{cdcl}_{NOT}\text{-merged-bj-learn } S T$
shows $(\text{not-simplified-cls } (\text{clauses } T)) \subseteq \# (\text{not-simplified-cls } (\text{clauses } S))$
using *assms* **apply** *induction*
prefer 4
unfolding *not-simplified-cls-def* **apply** (*auto elim!: backjump-lE forgetE*)[3]
by (*elim backjump-lE*) *auto*

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$:
assumes $\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T$
shows $(\text{not-simplified-cls } (\text{clauses } T)) \subseteq \# (\text{not-simplified-cls } (\text{clauses } S))$
using *assms* **apply** *induction*
apply *simp*
by (*drule cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*) *auto*

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-merged-bj-learn-clauses-bound}$:

assumes
 $cdcl_{NOT}$ -merged-bj-learn** S T **and**
 inv S **and**
 $atms$ -of- msu ($clauses$ S) \subseteq $atms$ -of- ms A **and**
 atm -of ‘($lits$ -of ($trail$ S)) \subseteq $atms$ -of- ms 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- ms 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** $cdcl_{NOT} = this(2)$ **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** $finite$ - cls - $S = this(7)$
have st' : $cdcl_{NOT}$ ** S T
using inv $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$ -and- inv st n - d **by** $blast$
have inv T
using inv $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn- inv st n - d **by** $blast$
moreover
have $atms$ -of- msu ($clauses$ T) \subseteq $atms$ -of- ms A **and**
 atm -of ‘ $lits$ -of ($trail$ T) \subseteq $atms$ -of- ms A
using $cdcl_{NOT}$. $rtranclp$ - $cdcl_{NOT}$ -trail- $clauses$ -bound[OF st'] inv $atms$ - $clss$ - S $atms$ - $trail$ - S n - d
by $blast+$
moreover moreover have no -dup ($trail$ T)
using $cdcl_{NOT}$. $rtranclp$ - $cdcl_{NOT}$ -no-dup[OF ‘ $cdcl_{NOT}$ ** 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- ms A)
using $cdcl_{NOT}$ $finite$ $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound
by ($auto$ $intro!$: $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound)
moreover have set - $mset$ (not -simplified- cls ($clauses$ T))
 \subseteq set - $mset$ (not -simplified- cls ($clauses$ S))
using $rtranclp$ - $cdcl_{NOT}$ -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 μ_{CDCL}' -bound **where**
 μ_{CDCL}' -bound A $T == ((2 + card$ ($atms$ -of- ms A)) \wedge ($1 + card$ ($atms$ -of- ms A))) \ast 2
 $+ card$ (set - $mset$ (not -simplified- cls ($clauses$ T)))
 $+ 3 \wedge card$ ($atms$ -of- ms A)

lemma $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound-card:

assumes
 $cdcl_{NOT}$ -merged-bj-learn** S T **and**
 inv S **and**
 $atms$ -of- msu ($clauses$ S) \subseteq $atms$ -of- ms A **and**
 atm -of ‘($lits$ -of ($trail$ S)) \subseteq $atms$ -of- ms A **and**
 n - d : no -dup ($trail$ S) **and**
 $finite$: $finite$ A
shows μ_{CDCL}' -merged A $T \leq \mu_{CDCL}'$ -bound A S
proof –
have set - $mset$ ($clauses$ T) \subseteq set - $mset$ (not -simplified- cls ($clauses$ S))
 \cup $build$ -all-simple- $clss$ ($atms$ -of- ms A)

using *rtrancp-cdcl_{NOT}-merged-bj-learn-clauses-bound*[*OF assms*] .
moreover have *card (set-mset (not-simplified-cls(clauses S))*
 \cup *build-all-simple-clss (atms-of-ms A))*
 \leq *card (set-mset (not-simplified-cls(clauses S))) + 3 ^ card (atms-of-ms A)*
by (*meson Nat.le-trans atms-of-ms-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 ^ card (atms-of-ms A)*
by (*meson build-all-simple-clss-finite card-mono dual-order.trans finite-UnI finite-set-mset*)
moreover have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T) * 2$
 $\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) * 2$
by *auto*
ultimately show *?thesis unfolding μ_{CDCL}' -merged-def* **by** *auto*
qed

sublocale *cdcl_{NOT}-increasing-restarts-ops* $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}] S$
cdcl_{NOT}-merged-bj-learn f
 $\lambda A S. \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged}$
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply *unfold-locales*
using *unbounded apply simp*
using *f-ge-1 apply force*
apply (*blast dest!: cdcl_{NOT}-merged-bj-learn-is-trancp-cdcl_{NOT} trancp-into-rtrancp*
cdcl_{NOT}.rtrancp-cdcl_{NOT}-trail-clauses-bound)
apply (*simp add: cdcl_{NOT}-decreasing-measure'*)
using *rtrancp-cdcl_{NOT}-merged-bj-learn-clauses-bound-card* **apply** *blast*
apply (*drule rtrancp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)
apply (*auto dest!: simp: card-mono set-mset-mono*)[]
apply *simp*
apply *auto*[]
using *cdcl_{NOT}-merged-bj-learn-no-dup-inv cdcl-merged-inv* **apply** *blast*
apply (*auto simp: inv-restart*)[]
done

lemma *cdcl_{NOT}-restart- μ_{CDCL}' -merged-le- μ_{CDCL}' -bound:*

assumes
cdcl_{NOT}-restart T V
inv (fst T) and
no-dup (trail (fst T)) and
atms-of-msu (clauses (fst T)) \subseteq atms-of-ms A and
atm-of ' lits-of (trail (fst T)) \subseteq atms-of-ms A and
finite A
shows $\mu_{CDCL}'\text{-merged } A \text{ (fst } V) \leq \mu_{CDCL}'\text{-bound } A \text{ (fst } T)$
using *assms*
proof *induction*
case (*restart-full S T n*)
show *?case*
unfolding *fst-conv*
apply (*rule rtrancp-cdcl_{NOT}-merged-bj-learn-clauses-bound-card*)
using *restart-full unfolding full1-def* **by** (*force dest!: trancp-into-rtrancp*) +
next
case (*restart-step m S T n U*) **note** *st = this(1) and U = this(3) and inv = this(4) and*

$n-d = \text{this}(5)$ **and** $\text{atms-clss} = \text{this}(6)$ **and** $\text{atms-trail} = \text{this}(7)$ **and** $\text{finite} = \text{this}(8)$
then have st' : $\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T$
by (*blast dest: relpowp-imp-rtrancp*)
then have st'' : $\text{cdcl}_{NOT}^{**} S T$
using $\text{inv } n-d$ **apply** – **by** (*rule rtrancp-cdcl_{NOT}-merged-bj-learn-is-rtrancp-cdcl_{NOT}*) *auto*
have $\text{inv } T$
apply (*rule rtrancp-cdcl_{NOT}-merged-bj-learn-inv*)
using $\text{inv } st' n-d$ **by** *auto*
then have $\text{inv } U$
using U **by** (*auto simp: inv-restart*)
have $\text{atms-of-msu } (\text{clauses } T) \subseteq \text{atms-of-ms } A$
using $\text{cdcl}_{NOT}.\text{rtrancp-cdcl}_{NOT}\text{-trail-clauses-bound}[OF st']$ $\text{inv atms-clss atms-trail } n-d$
by *simp*
then have $\text{atms-of-msu } (\text{clauses } U) \subseteq \text{atms-of-ms } A$
using U **by** *simp*
have $\text{not-simplified-cls } (\text{clauses } U) \subseteq \# \text{ not-simplified-cls } (\text{clauses } T)$
using $\langle U \sim \text{reduce-trail-to}_{NOT} [] T \rangle$ **by** *auto*
moreover have $\text{not-simplified-cls } (\text{clauses } T) \subseteq \# \text{ not-simplified-cls } (\text{clauses } S)$
apply (*rule rtrancp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)
using $\langle (\text{cdcl}_{NOT}\text{-merged-bj-learn} \sim m) S T \rangle$ **by** (*auto dest!: relpowp-imp-rtrancp*)
ultimately have $U-S: \text{not-simplified-cls } (\text{clauses } U) \subseteq \# \text{ not-simplified-cls } (\text{clauses } S)$
by *auto*

have ($\text{set-mset } (\text{clauses } U)$)
 $\subseteq \text{set-mset } (\text{not-simplified-cls } (\text{clauses } U)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A)$
apply (*rule rtrancp-cdcl_{NOT}-merged-bj-learn-clauses-bound*)
apply *simp*
using $\langle \text{inv } U \rangle$ **apply** *simp*
using $\langle \text{atms-of-msu } (\text{clauses } U) \subseteq \text{atms-of-ms } A \rangle$ **apply** *simp*
using U **apply** *simp*
using U **apply** *simp*
using finite **apply** *simp*
done

then have $f1: \text{card } (\text{set-mset } (\text{clauses } U)) \leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses } U)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A))$
by (*meson build-all-simple-clss-finite card-mono finite-UnI finite-set-mset*)

moreover have $\text{set-mset } (\text{not-simplified-cls } (\text{clauses } U)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A)$
 $\subseteq \text{set-mset } (\text{not-simplified-cls } (\text{clauses } S)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A)$
using $U-S$ **by** *auto*

then have $f2:$
 $\text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses } U)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses } S)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A))$
by (*meson build-all-simple-clss-finite card-mono finite-UnI finite-set-mset*)

moreover have $\text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses } S)) \cup \text{build-all-simple-clss } (\text{atms-of-ms } A))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses } S))) + \text{card } (\text{build-all-simple-clss } (\text{atms-of-ms } A))$
using card-Un-le **by** *blast*

moreover have $\text{card } (\text{build-all-simple-clss } (\text{atms-of-ms } A)) \leq 3 \wedge \text{card } (\text{atms-of-ms } A)$
using $\text{atms-of-ms-finite build-all-simple-clss-card local.finite}$ **by** *blast*

ultimately have $\text{card } (\text{set-mset } (\text{clauses } U))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses } S))) + 3 \wedge \text{card } (\text{atms-of-ms } A)$
by *linarith*

then show $?case$ **unfolding** $\mu_{CDCL}'\text{-merged-def}$ **by** *auto*

qed

lemma $cdcl_{NOT}\text{-restart-}\mu_{CDCL}'\text{-bound-le-}\mu_{CDCL}'\text{-bound}$:

assumes

$cdcl_{NOT}\text{-restart } T \ V$ **and**

$no\text{-dup } (trail \ (fst \ T))$ **and**

$inv \ (fst \ T)$ **and**

fin : $finite \ A$

shows $\mu_{CDCL}'\text{-bound } A \ (fst \ V) \leq \mu_{CDCL}'\text{-bound } A \ (fst \ T)$

using $assms(1-3)$

proof *induction*

case $(restart\text{-full } S \ T \ n)$

have $not\text{-simplified-cls } (clauses \ T) \subseteq \# \ not\text{-simplified-cls } (clauses \ S)$

apply $(rule \ rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-not-simplified-decreasing})$

using $\langle full1 \ cdcl_{NOT}\text{-merged-bj-learn } S \ T \rangle$ **unfolding** $full1\text{-def}$

by $(auto \ dest: \ trancplp\text{-into-rtranclp})$

then show $?case$ **by** $(auto \ simp: \ card\text{-mono} \ set\text{-mset-mono})$

next

case $(restart\text{-step } m \ S \ T \ n \ U)$ **note** $st = this(1)$ **and** $U = this(3)$ **and** $n\text{-d} = this(4)$ **and** $inv = this(5)$

then have st' : $cdcl_{NOT}\text{-merged-bj-learn}^{**} \ S \ T$

by $(blast \ dest: \ relpowp\text{-imp-rtranclp})$

then have st'' : $cdcl_{NOT}^{**} \ S \ T$

using $inv \ n\text{-d}$ **apply** $-$ **by** $(rule \ rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-is-rtranclp-cdcl_{NOT}})$ **auto**

have $inv \ T$

apply $(rule \ rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-inv})$

using $inv \ st' \ n\text{-d}$ **by** $auto$

then have $inv \ U$

using U **by** $(auto \ simp: \ inv\text{-restart})$

have $not\text{-simplified-cls } (clauses \ U) \subseteq \# \ not\text{-simplified-cls } (clauses \ T)$

using $\langle U \sim reduce\text{-trail-to}_{NOT} \ [] \ T \rangle$ **by** $auto$

moreover have $not\text{-simplified-cls } (clauses \ T) \subseteq \# \ not\text{-simplified-cls } (clauses \ S)$

apply $(rule \ rtranclp\text{-}cdcl_{NOT}\text{-merged-bj-learn-not-simplified-decreasing})$

using $\langle (cdcl_{NOT}\text{-merged-bj-learn} \ \widetilde{\sim} \ m) \ S \ T \rangle$ **by** $(auto \ dest!: \ relpowp\text{-imp-rtranclp})$

ultimately have $U\text{-S}$: $not\text{-simplified-cls } (clauses \ U) \subseteq \# \ not\text{-simplified-cls } (clauses \ S)$

by $auto$

then show $?case$ **by** $(auto \ simp: \ card\text{-mono} \ set\text{-mset-mono})$

qed

sublocale $cdcl_{NOT}\text{-increasing-restarts} \ - \ - \ - \ - \ f \ \lambda S \ T. \ T \sim reduce\text{-trail-to}_{NOT} \ ([::'a \ list]) \ S$

$\lambda A \ S. \ atms\text{-of-msu } (clauses \ S) \subseteq atms\text{-of-ms } A$

$\wedge atm\text{-of } ' \ lits\text{-of } (trail \ S) \subseteq atms\text{-of-ms } A \wedge finite \ A$

$\mu_{CDCL}'\text{-merged } cdcl_{NOT}\text{-merged-bj-learn}$

$\lambda S. \ inv \ S \wedge no\text{-dup } (trail \ S)$

$\lambda A \ T. \ ((2 + card \ (atms\text{-of-ms } A)) \wedge (1 + card \ (atms\text{-of-ms } A))) * 2$

$+ card \ (set\text{-mset } (not\text{-simplified-cls}(clauses \ T)))$

$+ 3 \wedge card \ (atms\text{-of-ms } A)$

apply $unfold\text{-locales}$

using $cdcl_{NOT}\text{-restart-}\mu_{CDCL}'\text{-merged-le-}\mu_{CDCL}'\text{-bound}$ **apply** $force$

using $cdcl_{NOT}\text{-restart-}\mu_{CDCL}'\text{-bound-le-}\mu_{CDCL}'\text{-bound}$ **by** $fastforce$

lemma $cdcl_{NOT}\text{-restart-eq-sat-iff}$:

assumes

$cdcl_{NOT}\text{-restart } S \ T$ **and**

```

    no-dup (trail (fst S))
    inv (fst S)
  shows  $I \models_{\text{sextm}} \text{clauses } (fst S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses } (fst T)$ 
  using assms
proof (induction rule: cdclNOT-restart.induct)
  case (restart-full S T n)
  then have cdclNOT-merged-bj-learn** S T
    by (simp add: trancpl-into-rtrancpl full1-def)
  then show ?case
    using cdclNOT.rtrancpl-cdclNOT-bj-sat-ext-iff restart-full.prem(1,2)
    rtrancpl-cdclNOT-merged-bj-learn-is-rtrancpl-cdclNOT by auto
next
  case (restart-step m S T n U)
  then have cdclNOT-merged-bj-learn** S T
    by (auto simp: trancpl-into-rtrancpl full1-def dest!: relpowp-imp-rtrancpl)
  then have  $I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$ 
    using cdclNOT.rtrancpl-cdclNOT-bj-sat-ext-iff restart-step.prem(1,2)
    rtrancpl-cdclNOT-merged-bj-learn-is-rtrancpl-cdclNOT by auto
  moreover have  $I \models_{\text{sextm}} \text{clauses } T \longleftrightarrow I \models_{\text{sextm}} \text{clauses } U$ 
    using restart-step.hyps(3) by auto
  ultimately show ?case by auto
qed

lemma rtrancpl-cdclNOT-restart-eq-sat-iff:
  assumes
    cdclNOT-restart** S T and
    inv: inv (fst S) and n-d: no-dup(trail (fst S))
  shows  $I \models_{\text{sextm}} \text{clauses } (fst S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses } (fst T)$ 
  using assms(1)
proof (induction rule: rtrancpl-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 rtrancpl-cdclNOT-with-restart-cdclNOT-inv using st inv n-d by blast+
  then have  $I \models_{\text{sextm}} \text{clauses } (fst T) \longleftrightarrow I \models_{\text{sextm}} \text{clauses } (fst U)$ 
    using cdclNOT-restart-eq-sat-iff cdcl by blast
  then show ?case using IH by blast
qed

lemma cdclNOT-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: trancpl-into-rtrancpl)+
have st': cdclNOT** S T
    using inv rtrancpl-cdclNOT-merged-bj-learn-is-rtrancpl-cdclNOT-and-inv st n-d by auto
have inv T
    using rtrancpl-cdclNOT-cdclNOT-inv[OF st] inv n-d by auto
then show ?case
    using cdclNOT.rtrancpl-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 *rtrancpl-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
  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 rtrancpl-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_{NOT}-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-cls: atms-of-msu (clauses (fst S)) ⊆ atms-of-ms A and
  atms-trail: atm-of ' lits-of (trail (fst S)) ⊆ atms-of-ms A and
  fin: finite A
shows unsatisfiable (set-mset (clauses (fst S)))
  ∨ lits-of (trail (fst T)) ⊨ sextm clauses (fst S) ∧ satisfiable (set-mset (clauses (fst S)))
proof -
  have inv-T: inv (fst T) and n-d-T: no-dup (trail (fst T))
    using rtrancpl-cdclNOT-with-restart-cdclNOT-inv using full inv n-d unfolding full-def by blast+
  moreover have
    atms-cls-T: atms-of-msu (clauses (fst T)) ⊆ atms-of-ms A and
    atms-trail-T: atm-of ' lits-of (trail (fst T)) ⊆ atms-of-ms A
  using rtrancpl-cdclNOT-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 cdclNOT-merged-bj-learn (fst T)
  apply -

```

```

apply (rule no-step-cdclNOT-restart-no-step-cdclNOT[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 rtrancp-cdclNOT-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)))
  ∨ trail (fst T) ⊨asm clauses (fst T) ∧ satisfiable (set-mset (clauses (fst T)))
  apply -
  apply (rule cdclNOT-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) ⊨asm clauses (fst T) and satisfiable (set-mset (clauses (fst T)))
  by auto
then show unsatisfiable (set-mset (clauses (fst S)))
  ∨ lits-of (trail (fst T)) ⊨sextm clauses (fst S) ∧ satisfiable (set-mset (clauses (fst S)))
proof cases
  case unsat
  then have unsatisfiable (set-mset (clauses (fst S)))
    unfolding satisfiable-def apply auto
    using rtrancp-cdclNOT-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)) ⊨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)) ⊨sextm clauses (fst S)
    using rtrancp-cdclNOT-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-cdclNOT-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 \times 'v clauses

\Rightarrow ('v, unit, unit) marked-lit list \times 'v clauses \Rightarrow bool **where**

backtrack-split (*fst* *S*) = (*M'*, *L* # *M*) \Longrightarrow *is-marked* *L* \Longrightarrow *D* \in # *snd* *S*

\Longrightarrow *fst* *S* \models_{as} *CNot* *D* \Longrightarrow *backtrack* *S* (*Propagated* ($-(\text{lit-of } L)$) () # *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* \circ *fst*) (*M*, *N*) **and**

decomp: *all-decomposition-implies-m* *N* (*get-all-marked-decomposition* *M*)

shows

$\exists C F' K F L l C'$.

$M = F' @ \text{Marked } K () \# F \wedge$

$M' = \text{Propagated } L l \# F \wedge N = N' \wedge C \in \# N \wedge F' @ \text{Marked } K d \# F \models_{as} \text{CNot } C \wedge$

undefined-lit *F* *L* \wedge *atm-of* *L* \in *atms-of-msu* *N* \cup *atm-of* ' *lits-of* (*F'* @ *Marked* *K* *d* # *F*) \wedge

$N \models_{pm} C' + \{\#L\} \wedge F \models_{as} \text{CNot } C'$

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 \in # *snd* ?*S* **and**

M $\models_{as} \text{CNot } D$ **and**

bt: *backtrack* ?*S* (*Propagated* ($-(\text{lit-of } L)$) *P* # *F*, *N*) **and**

M': *M'* = *Propagated* ($-(\text{lit-of } L)$) *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* { $\#K \in \#mset$ *M*. *is-marked* *K* \wedge *K* \neq *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'* @ *Marked* *K* () # *F* $\models_{as} \text{CNot } D$

using ' *M* $\models_{as} \text{CNot } D$ **unfolding** *M* .

moreover **have** *undefined-lit* *F* ($-(?K)$)

using *no-dup* **unfolding** *M* *L* **by** (*simp add: defined-lit-map*)

moreover **have** *atm-of* ($-K$) \in *atms-of-msu* *N* \cup *atm-of* ' *lits-of* (*F'* @ *Marked* *K* *d* # *F*)

by *auto*

moreover

have *set-mset* *N* \cup ?*C'* \models_{ps} { $\{\# \}$ }

proof –

have *A*: *set-mset* *N* \cup ?*C'* \cup ($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* *M* =

set-mset *N* \cup ($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* *M*

unfolding *M* *L* **by** *auto*

have *set-mset* *N* \cup { $\{\# \text{lit-of } L \# \}$ | *L*. *is-marked* *L* \wedge *L* \in *set* *M*}

```

     $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
  moreover have  $C'$ :  $?C' = \{\{\#lit\text{-of } L\# \} \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}$ 
    unfolding  $M$   $L$  apply standard
    apply force
    using IntI by auto
  ultimately have  $N\text{-}C\text{-}M$ :  $\text{set-mset } N \cup ?C' \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } M$ 
    by auto
  have  $\text{set-mset } N \cup (\lambda L. \{\#lit\text{-of } L\# \}) \text{ ' (set } M) \models_{ps} \{\{\# \}\}$ 
    unfolding true-clss-clss-def
  proof (intro allI impI, goal-cases)
    case (1  $I$ ) note  $tot = \text{this}(1)$  and  $cons = \text{this}(2)$  and  $I\text{-}N\text{-}M = \text{this}(3)$ 
    have  $I \models D$ 
      using  $I\text{-}N\text{-}M \langle D \in \# \text{ snd } ?S \rangle$  unfolding true-clss-def by auto
    moreover have  $I \models_s CNot\ D$ 
      using  $\langle M \models_{as} CNot\ D \rangle$  unfolding  $M$  by (metis 1(3)  $\langle M \models_{as} CNot\ D \rangle$ 
        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  $N \models_{pm} \text{image-mset } uminus\ ?C + \{\# - ?K\# \}$ 
    unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
  proof (intro allI impI)
    fix  $I$ 
    assume
       $tot$ :  $\text{total-over-set } I (\text{atms-of-ms } (\text{set-mset } N \cup \{\text{image-mset } uminus\ ?C + \{\# - ?K\# \}\}))$  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  $a2$ :  $\forall l \in \text{atm-of ' lit-of ' (set } M \cap \{L. \text{ is-marked } L \wedge L \neq \text{Marked } K\ d\}).$ 
       $Pos\ l \in I \vee Neg\ l \in I$ 
    assume  $a3$ :  $\text{lit-of } x \notin I$ 
    assume  $a4$ :  $\text{is-marked } x$ 
    assume  $a5$ :  $x \neq \text{Marked } K\ d$ 
    have  $f6$ :  $Neg (\text{atm-of } (\text{lit-of } x)) = - Pos (\text{atm-of } (\text{lit-of } x))$ 
      by simp
    have  $Pos (\text{atm-of } (\text{lit-of } x)) \in I \vee Neg (\text{atm-of } (\text{lit-of } x)) \in I$ 
      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)
  end

```



```

      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-ms-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  $\langle (K \in I \wedge \neg K \notin I) \vee (\neg K \in I \wedge K \notin I) \rangle$ 
    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 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  $M M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$ 
  assumes
    backtrack:  $\text{backtrack } S T$  and
    no-dup:  $(\text{no-dup} \circ \text{fst}) S$  and
    decomp:  $\text{all-decomposition-implies-m } (\text{snd } S) (\text{get-all-marked-decomposition } (\text{fst } S))$ 
  shows
     $\exists C F' K F L l C'.$ 
     $\text{fst } S = F' @ \text{Marked } K () \# F \wedge$ 
     $T = (\text{Propagated } L l \# F, \text{snd } S) \wedge C \in \# \text{ snd } S \wedge \text{fst } S \models_{as} \text{CNot } C$ 
     $\wedge \text{undefined-lit } F L \wedge \text{atm-of } L \in \text{atms-of-msu } (\text{snd } S) \cup \text{atm-of ' lits-of } (\text{fst } S) \wedge$ 
     $\text{snd } S \models_{pm} C' + \{\#L\#\} \wedge F \models_{as} \text{CNot } C'$ 
  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  $\lambda L (M, N). (L \# M, N) \lambda(M, N). (\text{tl } M, N)$ 
 $\lambda C (M, N). (M, \{\#C\#\} + N) \lambda C (M, N). (M, \text{remove-mset } C N)$ 
by unfold-locales auto

sublocale backjumping-ops fst snd  $\lambda L (M, N). (L \# M, N) \lambda(M, N). (\text{tl } M, N)$ 
 $\lambda C (M, N). (M, \{\#C\#\} + N) \lambda C (M, N). (M, \text{remove-mset } C N) \lambda - - S T. \text{backtrack } S T$ 
by unfold-locales

lemma backtrack-is-backjump'':
  fixes  $M M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$ 
  assumes
    backtrack:  $\text{backtrack } S T$  and
    no-dup:  $(\text{no-dup} \circ \text{fst}) S$  and
    decomp:  $\text{all-decomposition-implies-m } (\text{snd } S) (\text{get-all-marked-decomposition } (\text{fst } S))$ 
  shows  $\text{backjump } S T$ 
proof -
  obtain  $C F' K F L l C'$  where
    1:  $\text{fst } S = F' @ \text{Marked } K () \# F$  and
    2:  $T = (\text{Propagated } L l \# F, \text{snd } S)$  and
    3:  $C \in \# \text{ snd } S$  and

```

```

4:  $\text{fst } S \models_{\text{as}} \text{CNot } C$  and
5: undefined-lit  $F \ L$  and
6:  $\text{atm-of } L \in \text{atms-of-msu } (\text{snd } S) \cup \text{atm-of ' lits-of } (\text{fst } S)$  and
7:  $\text{snd } S \models_{\text{pm}} C' + \{\#L\# \}$  and
8:  $F \models_{\text{as}} \text{CNot } C'$ 
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-eqNOT-def simp del: state-simpNOT)
qed

lemma can-do-bt-step:
  assumes
     $M: \text{fst } S = F' @ \text{Marked } K \ d \ \# \ F$  and
     $C \in \# \text{snd } S$  and
     $C: \text{fst } S \models_{\text{as}} \text{CNot } C$ 
  shows  $\neg \text{no-step backtrack } S$ 
proof -
  obtain  $L \ G' \ G$  where
    backtrack-split  $(\text{fst } S) = (G', L \# G)$ 
  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  $S \ G' \ L \ G \ C$ ]  $\langle C \in \# \text{snd } S \rangle$  C unfolding M by auto
qed

end

sublocale dpll-with-backtrack  $\subseteq$  dpll-with-backjumping-ops fst snd  $\lambda L \ (M, N). (L \# M, N)$ 
 $\lambda(M, N). (\text{tl } M, N) \lambda C \ (M, N). (M, \{\#C\# \} + N) \lambda C \ (M, N). (M, \text{remove-mset } C \ N) \lambda - -. \text{True}$ 
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N \ (\text{get-all-marked-decomposition } M)$ 
 $\lambda - - S \ T. \text{backtrack } S \ T$ 
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  $\lambda L \ (M, N). (L \# M, N)$ 
 $\lambda(M, N). (\text{tl } M, N) \lambda C \ (M, N). (M, \{\#C\# \} + N) \lambda C \ (M, N). (M, \text{remove-mset } C \ N) \lambda - -. \text{True}$ 
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N \ (\text{get-all-marked-decomposition } M)$ 
 $\lambda - - S \ T. \text{backtrack } S \ T$ 
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
 $\text{fst snd } \lambda L \ (M, N). (L \# M, N)$ 
 $\lambda(M, N). (\text{tl } M, N) \lambda C \ (M, N). (M, \{\#C\# \} + N) \lambda C \ (M, N). (M, \text{remove-mset } C \ N) \lambda - -. \text{True}$ 
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N \ (\text{get-all-marked-decomposition } M)$ 
 $\lambda - - S \ T. \text{backtrack } S \ T \lambda - -. \text{False } \lambda - -. \text{False}$ 
by unfold-locales

sublocale dpll-with-backtrack  $\subseteq$  conflict-driven-clause-learning
 $\text{fst snd } \lambda L \ (M, N). (L \# M, N)$ 
 $\lambda(M, N). (\text{tl } M, N) \lambda C \ (M, N). (M, \{\#C\# \} + N) \lambda C \ (M, N). (M, \text{remove-mset } C \ N) \lambda - -. \text{True}$ 
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N \ (\text{get-all-marked-decomposition } M)$ 

```

```

λ- - S T. backtrack S T λ- -. False λ- -. False
apply unfold-locales
using cdclNOT.simps dpll-bj-inv forgetE learnE by blast

context dpll-with-backtrack
begin
lemma wf-tranclp-dpll-inital-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') ∧ atms-of-msu N ⊆ atms-of-ms 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) ∨ (M' ⊨asm N ∧ 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' ⊨asm N ⟷ 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' ⊨asm N ⟹ 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 cdclNOT-is-dpll:
  cdclNOT S T ⟷ dpll-bj S T
  by (auto simp: cdclNOT.simps learn.simps forgetNOT.simps)

Another proof of termination:
lemma wf {(T, S). dpll-bj S T ∧ cdclNOT-NOT-all-inv A S}
  unfolding cdclNOT-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 ⇒ nat
  assumes unbounded: unbounded f and f-ge-1: ∧ n. n ≥ 1 ⟹ f n ≥ 1
begin
  sublocale cdclNOT-increasing-restarts fst snd λL (M, N). (L # M, N) λ(M, N). (tl M, N)
    λC (M, N). (M, {#C#} + N) λC (M, N). (M, remove-mset C N) f λ(-, N) S. S = ([], N)
  λA (M, N). atms-of-msu N ⊆ atms-of-ms A ∧ atm-of ' lits-of M ⊆ atms-of-ms A ∧ finite A
    ∧ all-decomposition-implies-m N (get-all-marked-decomposition M)

```

```

λA T. (2+card (atms-of-ms A)) ^ (1+card (atms-of-ms A))
      - μC (1+card (atms-of-ms A)) (2+card (atms-of-ms A)) (trail-weight T) dpll-bj
λ(M, N). no-dup M ∧ all-decomposition-implies-m N (get-all-marked-decomposition M)
λA -. (2+card (atms-of-ms A)) ^ (1+card (atms-of-ms 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 × 'v clauses

```

```

abbreviation trail :: 'v dpllW-state ⇒ 'v dpllW-marked-lits where
  trail ≡ fst
abbreviation clauses :: 'v dpllW-state ⇒ 'v clauses where
  clauses ≡ snd

```

The definition of DPLL is given in figure 2.13 page 70 of CW.

```

inductive dpllW :: 'v dpllW-state ⇒ 'v dpllW-state ⇒ bool where
  propagate: C + {#L#} ∈ # clauses S ⇒ trail S ⊨as CNot C ⇒ undefined-lit (trail S) L
    ⇒ dpllW S (Propagated L () # trail S, clauses S) |
  decided: undefined-lit (trail S) L ⇒ atm-of L ∈ atms-of-msu (clauses S)
    ⇒ dpllW S (Marked L () # trail S, clauses S) |
  backtrack: backtrack-split (trail S) = (M', L # M) ⇒ is-marked L ⇒ D ∈ # clauses S
    ⇒ trail S ⊨as CNot D ⇒ dpllW S (Propagated (- (lit-of L)) () # M, clauses S)

```

16.2 Invariants

```

lemma dpllW-distinct-inv:
  assumes dpllW S S'
  and no-dup (trail S)
  shows no-dup (trail S')
  using assms
proof (induct rule: dpllW.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_W-consistent-interp-inv*:
 assumes *dpll_W 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_W.induct*)
 case (backtrack $S M' L M D$) note extracted = this(1) and marked = this(2) and $D = \text{this}(4)$ 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) \subseteq 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 $L \notin$ 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 ($-\text{lit-of } L$) \notin ($\lambda m. \text{atm-of (lit-of } m)$) ' set M
 using no-dup backtrack-split-list-eq[of trail S , symmetric] unfolding extracted by force
 then have $-\text{lit-of } L \notin$ lits-of M
 unfolding lits-of-def by force
 ultimately show ?case by simp
 qed (*auto intro: consistent-add-undefined-lit-consistent*)

lemma *dpll_W-vars-in-snd-inv*:
 assumes *dpll_W S S'*
 and atm-of ' (lits-of (trail S)) \subseteq atms-of-msu (clauses S)
 shows atm-of ' (lits-of (trail S')) \subseteq atms-of-msu (clauses S')
 using *assms*
proof (*induct rule: dpll_W.induct*)
 case (backtrack $S M' L M D$)
 then have atm-of (lit-of L) \in atms-of-msu (clauses S)
 using backtrack-split-list-eq[of trail S , symmetric] by auto
 moreover
 have atm-of ' lits-of (trail S) \subseteq atms-of-msu (clauses S)
 using backtrack(5) by simp
 then have $\bigwedge x b. x b \in \text{set } M \implies \text{atm-of (lit-of } x b) \in \text{atms-of-msu (clauses } S)$
 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-ms*)

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms (($\lambda a. \{\# \text{lit-of } a \# \}$) ' c) = atm-of ' lit-of ' c*
 unfolding atms-of-ms-def using image-iff by force

lemma *dpll_W-propagate-is-conclusion:*

assumes *dpll_W S S'*

and *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

and *atm-of 'lits-of (trail S) ⊆ atms-of-msu (clauses S)*

shows *all-decomposition-implies-m (clauses S') (get-all-marked-decomposition (trail S'))*

using *assms*

proof (*induct rule: dpll_W.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 ⊨_p C + {#L#}* **by** (*auto simp add: inS*)

moreover have *?I ⊨_{ps} CNot C* **using** *true-annots-true-clss-cls cnot* **by** *fastforce*

ultimately have *?I ⊨_p {#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)))*

$\implies ((\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } a \cup \text{set-mset (clauses S)}) \models_{ps} (\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } b$

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)*

$\implies ((\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } a \cup \text{set-mset (clauses S)}) \models_{ps} ((\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } c)$

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)*

$\implies ((\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } a \cup \text{set-mset (clauses S)}) \models_p \{ \# L \# \}$

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)*

$\cup (\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } c \models_{ps} \text{CNot } C$

using *?I ⊨_{ps} CNot C* **unfolding** *h'* **by** (*simp add: Un-commute Un-left-commute*)

have

$\text{atms-of-ms (CNot } C) \subseteq \text{atms-of-ms (set (map (λa. \{ \# \text{lit-of } a \# \}) a) \cup \text{set-mset (clauses S)})}$

and

$\text{atms-of-ms ((λa. \{ \# \text{lit-of } a \# \}) \text{ ' set } c) \subseteq \text{atms-of-ms (set (map (λa. \{ \# \text{lit-of } a \# \}) a) \cup \text{set-mset (clauses S)})}$

$\cup \text{set-mset (clauses S)}$

apply (*metis CNot-plus Un-subset-iff atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of*

atms-of-ms-union inS mem-set-mset-iff sup.coboundedI2)

using *inS atms-of-atms-of-ms-mono atms-incl* **by** (*fastforce simp: h'*)

then have $(\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } a \cup \text{set-mset (clauses S)} \models_{ps} \text{CNot } C$

using *true-clss-clss-left-right[OF - I] h 2* **by** *auto*

then show $(\lambda a. \{ \# \text{lit-of } a \# \}) \text{ ' set } a \cup \text{set-mset (clauses S)} \models_p \{ \# L \# \}$

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':  $\forall l \in \text{set } M'. \neg \text{is-marked } l$ 
  using extracted backtrack-split-fst-not-marked[of - trail S] by simp
have n: get-all-marked-decomposition (trail S)  $\neq []$  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:  $(\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } (L \# M) \cup \text{set-mset } (\text{clauses } S) \models_{ps} (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } M'$ 
  by simp
moreover
have  $(\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } (L \# M) \cup (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } M' \models_{ps} \text{CNot } D$ 
  by (metis (mono-tags, lifting) S Un-commute cons image-Un set-append
    true-annots-true-clss-clss)
then have 2:  $(\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } (L \# M) \cup \text{set-mset } (\text{clauses } S) \cup (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } M'$ 
   $\models_{ps} \text{CNot } D$ 
  by (metis (no-types, lifting) Un-assoc Un-left-commute true-clss-clss-union-l-r)
ultimately
have set (map  $(\lambda a. \{\# \text{lit-of } a\# \}) (L \# M)$ )  $\cup \text{set-mset } (\text{clauses } S) \models_{ps} \text{CNot } D$ 
  using true-clss-clss-left-right by fastforce
then have set (map  $(\lambda a. \{\# \text{lit-of } a\# \}) (L \# M)$ )  $\cup \text{set-mset } (\text{clauses } S) \models_p \{\#\}$ 
  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:  $(\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } M \cup \text{set-mset } (\text{clauses } S) \models_p \{\# - \text{lit-of } L\# \}$ 
  using true-clss-clss-false-left-right by auto
show ?case unfolding S all-decomposition-implies-def
proof
  fix x P level
  assume x:  $x \in \text{set } (\text{get-all-marked-decomposition } (\text{fst } (\text{Propagated } (- \text{lit-of } L) P \# M, \text{clauses } S)))$ 
  let ?M' =  $\text{Propagated } (- \text{lit-of } L) P \# M$ 
  let ?hd =  $\text{hd } (\text{get-all-marked-decomposition } ?M')$ 
  let ?tl =  $\text{tl } (\text{get-all-marked-decomposition } ?M')$ 
  have x = ?hd  $\vee x \in \text{set } ?tl$ 
  using x
  by (cases get-all-marked-decomposition ?M')
    auto
  moreover {
    assume x':  $x \in \text{set } ?tl$ 
    have L':  $\text{Marked } (\text{lit-of } L) () = L$  using marked by (case-tac L, auto)
    have x  $\in \text{set } (\text{get-all-marked-decomposition } (M' @ L \# M))$ 
      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)  $\Rightarrow (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set } Ls \cup \text{set-mset } (\text{clauses } S) \models_{ps} (\lambda a. \{\# \text{lit-of } a\# \}) \text{ ' set seen}$ 
      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  $f1$ :  $\bigwedge ms. \text{length} \ (get-all-marked-decomposition \ (M' @ ms))$ 
       $= \text{length} \ (get-all-marked-decomposition \ ms)$ 
    by (simp add:  $M' \ get-all-marked-decomposition-remove-unmarked-length$ )
    have  $Suc$  ( $\text{length} \ (get-all-marked-decomposition \ M) \neq Suc \ 0$ )
    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  $x'': x = (M0, \text{Propagated} \ (-lit-of \ L) \ P \# M0')$ 
  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 $dpll_W$ -propagate-is-conclusion-of-decided:

assumes $dpll_W \ S \ S'$
 and $all-decomposition-implies-m \ (clauses \ S) \ (get-all-marked-decomposition \ (trail \ S))$
 and $atm-of \ ' \ lits-of \ (trail \ S) \subseteq \ atms-of-msu \ (clauses \ S)$
 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 $all-decomposition-implies-trail-is-implied[OF \ dpll_W$ -propagate-is-conclusion $[OF \ assms]]$.

Lemma theorem 2.8.4 page 72 of CW

lemma *only-propagated-vars-unsat*:

assumes *marked*: $\forall x \in \text{set } M. \neg \text{is-marked } x$
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* $M \subseteq \text{atms-of-ms } N$
shows *unsatisfiable* N
proof (*rule ccontr*)
assume $\neg \text{unsatisfiable } N$
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 $l0: \{\{\#lit\text{-of } L\# \mid L. \text{is-marked } L \wedge L \in \text{set } M\} = \{\}\}$ **using** *marked* **by** *auto*
have *atms-of-ms* $(N \cup (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ‘ set } M) = \text{atms-of-ms } N$
using *atm-incl* **unfolding** *atms-of-ms-def lits-of-def* **by** *auto*

then have *total-over-m* $I\ (N \cup (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ‘ (set } M))$
using *tot* **unfolding** *total-over-m-def* **by** *auto*
then have $I \models_s (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ‘ (set } M)$
using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons* I
unfolding *true-clss-clss-def l0* **by** *auto*
then have $IM: I \models_s (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ‘ set } M$ **by** *auto*
{
fix K
assume $K \in \# D$
then have $\neg K \in \text{lits-of } M$
by (*auto split: split-if-asm*
intro: allE[*OF* $D[\text{unfolded true-annots-def Ball-def}], \text{ of } \{\# \neg K\# \}$])
then have $\neg K \in I$ **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_W-same-clauses*:

assumes *dpll_W* $S\ S'$
shows *clauses* $S = \text{clauses } S'$
using *assms* **by** (*induct rule: dpll_W.induct, auto*)

lemma *rtrancpl-dpll_W-inv*:

assumes *rtrancpl dpll_W* $S\ S'$
and *inv*: *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-decomposition* (*trail* S))
and *atm-incl*: *atm-of* ‘*lits-of* (*trail* S) $\subseteq \text{atms-of-msu}$ (*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') $\subseteq \text{atms-of-msu}$ (*clauses* S')
and *clauses* $S = \text{clauses } S'$
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-msu (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-msu (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-msu (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-msu (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-msu (clauses S)
   $\wedge$  consistent-interp (lits-of (trail S))
   $\wedge$  no-dup (trail S))

lemma dpllW-all-inv-dest[dest]:
  assumes dpllW-all-inv S
  shows all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
  and atm-of ‘ lits-of (trail S)  $\subseteq$  atms-of-msu (clauses S)
  and consistent-interp (lits-of (trail S))  $\wedge$  no-dup (trail S)
  using assms unfolding dpllW-all-inv-def lits-of-def by auto

lemma rtranclp-dpllW-all-inv:
  assumes rtranclp dpllW S S'
  and dpllW-all-inv S
  shows dpllW-all-inv S'
  using assms rtranclp-dpllW-inv[OF assms(1)] unfolding dpllW-all-inv-def lits-of-def by blast

lemma dpllW-all-inv:
  assumes dpllW S S'
  and dpllW-all-inv S

```

shows $dpll_W\text{-all-inv } S'$
using $assms \text{ rtrancpl-dpll}_W\text{-all-inv}$ **by** *blast*

lemma $rtrancpl\text{-dpll}_W\text{-inv-starting-from-0}$:

assumes $rtrancpl \text{ dpll}_W S S'$

and inv : $trail S = []$

shows $dpll_W\text{-all-inv } S'$

proof –

have $dpll_W\text{-all-inv } S$

using $assms \text{ unfolding all-decomposition-implies-def dpll}_W\text{-all-inv-def}$ **by** *auto*

then show $?thesis$ **using** $rtrancpl\text{-dpll}_W\text{-all-inv}[OF \text{ assms}(1)]$ **by** *blast*

qed

lemma $dpll_W\text{-can-do-step}$:

assumes $consistent\text{-interp } (set M)$

and $distinct M$

and $atm\text{-of } ' (set M) \subseteq atm\text{-of-msu } N$

shows $rtrancpl \text{ dpll}_W ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$

using $assms$

proof ($induct M$)

case *Nil*

then show $?case$ **by** *auto*

next

case ($Cons L M$)

then have $undefined\text{-lit } (map (\lambda M. \text{Marked } M ()) M) L$

unfolding $defined\text{-lit-def consistent-interp-def}$ **by** *auto*

moreover have $atm\text{-of } L \in atm\text{-of-msu } N$ **using** $Cons.prems(3)$ **by** *auto*

ultimately have $dpll_W (map (\lambda M. \text{Marked } M ()) M, N) (map (\lambda M. \text{Marked } M ()) (L \# M), N)$

using $dpll_W.decided$ **by** *auto*

moreover have $consistent\text{-interp } (set M)$ **and** $distinct M$ **and** $atm\text{-of } ' set M \subseteq atm\text{-of-msu } N$

using $Cons.prems \text{ unfolding consistent-interp-def}$ **by** *auto*

ultimately show $?case$ **using** $Cons.hyps$ **by** *auto*

qed

definition $conclusive\text{-dpll}_W\text{-state } (S:: 'v \text{ dpll}_W\text{-state}) \longleftrightarrow$

$(trail S \models_{asm} clauses S \vee ((\forall L \in set (trail S). \neg is\text{-marked } L)$

$\wedge (\exists C \in \# clauses S. trail S \models_{as} CNot C)))$

lemma $dpll_W\text{-strong-completeness}$:

assumes $set M \models_{sm} N$

and $consistent\text{-interp } (set M)$

and $distinct M$

and $atm\text{-of } ' (set M) \subseteq atm\text{-of-msu } N$

shows $dpll_W^{**} ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$

and $conclusive\text{-dpll}_W\text{-state } (map (\lambda M. \text{Marked } M ()) M, N)$

proof –

show $rtrancpl \text{ dpll}_W ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$ **using** $dpll_W\text{-can-do-step assms}$ **by** *auto*

have $map (\lambda M. \text{Marked } M ()) M \models_{asm} N$ **using** $assms(1) \text{ true-annots-marked-true-cl}$ **by** *auto*

then show $conclusive\text{-dpll}_W\text{-state } (map (\lambda M. \text{Marked } M ()) M, N)$

unfolding $conclusive\text{-dpll}_W\text{-state-def}$ **by** *auto*

qed

lemma $dpll_W\text{-sound}$:

```

assumes
  rtrancpl dpllW ([], N) (M, N) and
   $\forall S. \neg dpll_W (M, N) S$ 
shows  $M \models_{asm} N \longleftrightarrow \text{satisfiable } (set\text{-}mset\ N) \text{ (is } ?A \longleftrightarrow ?B)$ 
proof
  let  $?M' = lits\text{-}of\ M$ 
  assume  $?A$ 
  then have  $?M' \models_{sm} N$  by (simp add: true-annots-true-cl)
  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  $(\exists L. \text{undefined-lit } M\ L \wedge atm\text{-}of\ L \in atm\text{-}of\text{-}msu\ N) \vee (\exists D \in \#N. M \models_{as} CNot\ D)$ 
    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  $(\exists L. \text{undefined-lit } M\ L \wedge atm\text{-}of\ L \in atm\text{-}of\ D) \vee M \models_{as} CNot\ D$ 
      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-cl-def by blast
      then show  $?thesis$ 

      using  $D$  apply auto by (meson atm-of-atms-of-ms-mono mem-set-mset-iff subset-eq)
    qed
  moreover {
    assume  $\exists L. \text{undefined-lit } M\ L \wedge atm\text{-}of\ L \in atm\text{-}of\text{-}msu\ N$ 
    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  $\forall l \in set\ M. \neg is\text{-}marked\ l$ 
      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  $M\ D\ set\text{-}mset\ N$ ]  $DN\ MD$ 
        rtrancpl-dpllW-all-inv[OF assms(1)] by force
      then have False using  $\langle ?B \rangle$  by blast
    }
  }
  moreover {
    assume  $l: \exists l \in set\ M. is\text{-}marked\ l$ 
    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 $dpll_W\text{-mes } M \ n =$

$\text{map } (\lambda l. \text{ if is-marked } l \text{ then } 2 \text{ else } (1::\text{nat})) \ (\text{rev } M) \ @ \ \text{replicate } (n - \text{length } M) \ 3$

lemma $\text{length-dpll}_W\text{-mes}:$

assumes $\text{length } M \leq n$

shows $\text{length } (dpll_W\text{-mes } M \ n) = n$

using *assms unfolding dpll_W-mes-def by auto*

lemma $\text{distinctcard-atm-of-lit-of-eq-length}:$

assumes $\text{no-dup } S$

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 $dpll_W\text{-card-decrease}:$

assumes $dpll: dpll_W \ S \ S' \text{ and } \text{length } (\text{trail } S') \leq \text{card vars}$

and $\text{length } (\text{trail } S) \leq \text{card vars}$

shows $(dpll_W\text{-mes } (\text{trail } S') \ (\text{card vars}), dpll_W\text{-mes } (\text{trail } S) \ (\text{card vars}))$

$\in \text{lexn } \{(a, b). a < b\} \ (\text{card vars})$

using *assms*

proof (*induct rule: dpll_W.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_W-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[s] unfolding dpll_W-mes-def by (force simp add: lexn-conv assms(2))*

next

case (*backtrack S M' L M D*)

have $L: \text{is-marked } L$ **using** *backtrack.hyps(2) by auto*

have $S: \text{trail } S = M' @ L \# M$

using *backtrack.hyps(1) backtrack-split-list-eq[of trail S] by auto*

show *?case*

using *backtrack.prem[s] L unfolding dpll_W-mes-def S by (fastforce simp add: lexn-conv assms(2))*

qed

Proposition theorem 2.8.7 page 73 of CW

lemma $dpll_W\text{-card-decrease}':$

assumes $dpll: dpll_W \ S \ S'$

and $\text{atm-incl: atm-of } \text{' lits-of } (\text{trail } S) \subseteq \text{atms-of-msu } (\text{clauses } S)$

and $\text{no-dup: no-dup } (\text{trail } S)$

shows $(dpll_W\text{-mes } (\text{trail } S') \ (\text{card } (\text{atms-of-msu } (\text{clauses } S'))),$

$dpll_W\text{-mes } (\text{trail } S) \ (\text{card } (\text{atms-of-msu } (\text{clauses } S)))) \in \text{lex } \{(a, b). a < b\}$

proof —

```

have finite (atms-of-msu (clauses S)) unfolding atms-of-ms-def by auto
then have 1: length (trail S) ≤ card (atms-of-msu (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-msu (clauses S')
    using atm-incl dpll dpllW-vars-in-snd-inv[OF dpll] by force
  have finite (atms-of-msu (clauses S'))
    unfolding atms-of-ms-def by auto
  then have 2: length (trail S') ≤ card (atms-of-msu (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-msu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-msu (clauses S'))))
  ∈ lex {(a, b). a < b} (card (atms-of-msu (clauses S)))
  using dpllW-card-decrease[OF assms(1), of atms-of-msu (clauses S)] by blast
then have (dpllW-mes (trail S') (card (atms-of-msu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-msu (clauses S')))) ∈ lex {(a, b). a < b}
  unfolding lex-def by auto
then show (dpllW-mes (trail S') (card (atms-of-msu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-msu (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-msu (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 dpllW-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-msu (clauses S)))])
  using dpllW-card-decrease' by fast

lemma dpllW-trancpl-star-commute:
  {(S', S). dpllW-all-inv S ∧ dpllW S S'}+ = {(S', S). dpllW-all-inv S ∧ trancpl dpllW 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-trancl')
  next
    case (trancl-into-trancl S S' S'')
    then have  $(S', S) \in \{a. \text{case } a \text{ of } (S', S) \Rightarrow \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+$  by blast
    moreover have  $\text{dpll}_W\text{-all-inv } S'$ 
      using rtranclp-dpllW-all-inv[OF tranclp-into-rtranclp[OF trancl-into-trancl.hyps(1)]]
      trancl-into-trancl.prem by auto
    ultimately have  $(S'', S') \in \{(pa, p). \text{dpll}_W\text{-all-inv } p \wedge \text{dpll}_W p pa\}^+$ 
      using  $\langle \text{dpll}_W\text{-all-inv } S' \rangle \text{trancl-into-trancl.hyps}(3)$  by blast
    then show ?case
      using  $\langle (S', S) \in \{a. \text{case } a \text{ of } (S', S) \Rightarrow \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+ \rangle$  by auto
  qed
}
then show ?B  $\subseteq$  ?A by blast
qed

```

lemma *dpll_W-wf-tranclp*: $\text{wf } \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W^{++} S S'\}$
unfolding *dpll_W-tranclp-star-commute*[symmetric] **by** (simp add: dpll_W-wf wf-trancl)

lemma *dpll_W-wf-plus*:
shows $\text{wf } \{(S', ([], N)) \mid S'. \text{dpll}_W^{++} ([], N) S'\}$ (is wf ?P)
apply (rule wf-subset[OF dpll_W-wf-tranclp, of ?P])
using *assms* **unfolding** *dpll_W-all-inv-def* **by** auto

16.4 Final States

lemma *dpll_W-no-more-step-is-a-conclusive-state*:

assumes $\forall S'. \neg \text{dpll}_W S S'$

shows *conclusive-dpll_W-state* S

proof –

have *vars*: $\forall s \in \text{atms-of-msu } (\text{clauses } S). s \in \text{atm-of ' lits-of (trail } S)$

proof (rule ccontr)

assume $\neg (\forall s \in \text{atms-of-msu } (\text{clauses } S). s \in \text{atm-of ' lits-of (trail } S))$

then obtain L **where**

$L\text{-in-atms}$: $L \in \text{atms-of-msu } (\text{clauses } S)$ **and**

$L\text{-notin-trail}$: $L \notin \text{atm-of ' lits-of (trail } S)$ **by** *metis*

obtain L' **where** $L': \text{atm-of } L' = L$ **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** *dpll_W.decided assms*(1) $L\text{-in-atms } L'$ **by** blast

qed

show ?thesis

proof (rule ccontr)

assume *not-final*: $\neg ?thesis$

then have

$\neg \text{trail } S \models_{\text{asm}} \text{clauses } S$ **and**

$(\exists L \in \text{set } (\text{trail } S). \text{is-marked } L) \vee (\forall C \in \# \text{clauses } S. \neg \text{trail } S \models_{\text{as}} C \text{Not } C)$

unfolding *conclusive-dpll_W-state-def* **by** auto

moreover {

assume $\exists L \in \text{set } (\text{trail } S). \text{is-marked } L$

then obtain $L M' M$ **where** $L: \text{backtrack-split } (\text{trail } S) = (M', L \# M)$

using *backtrack-split-some-is-marked-then-snd-has-hd* **by** blast

obtain D **where** $D \in \# \text{clauses } S$ **and** $\neg \text{trail } S \models_a D$

using $\langle \neg \text{trail } S \models_{\text{asm}} \text{clauses } S \rangle$ **unfolding** *true-annots-def* **by** auto

then have $\forall s \in \text{atms-of-ms } \{D\}. s \in \text{atm-of ' lits-of (trail } S)$

using *vars* **unfolding** *atms-of-ms-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  $\langle D \in \# \text{ clauses } S \rangle \langle \text{trail } S \models_{as} \text{CNot } D \rangle$  by blast
  }
  moreover {
    assume tr:  $\forall C \in \# \text{ clauses } S. \neg \text{trail } S \models_{as} \text{CNot } C$ 
    obtain C where C-in-cl:  $C \in \# \text{ clauses } S$  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  $\forall s \in \text{atms-of-ms } \{C\}. s \in \text{atm-of ' lits-of } (\text{trail } S)$ 
      using vars  $\langle C \in \# \text{ clauses } S \rangle$  unfolding atms-of-ms-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** ( $\square$ , N) (M, N) and conclusive-dpllW-state (M, N)
  shows  $M \models_{asm} N \longleftrightarrow \text{satisfiable } (\text{set-mset } N)$  (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 rtrancp-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:  $\forall L \in \text{set } M. \neg \text{is-marked } L \exists C \in \# N. M \models_{as} \text{CNot } C$ 
      using n assms(2) unfolding conclusive-dpllW-state-def by auto
    moreover obtain D where DN:  $D \in \# N$  and MD:  $M \models_{as} \text{CNot } D$  using no-mark by auto
    ultimately have unsatisfiable (set-mset N)
      using only-propagated-vars-unsat rtrancp-dpllW-all-inv[OF assms(1)]
      unfolding dpllW-all-inv-def by force
    then show False using  $\langle ?B \rangle$  by blast
  qed
qed
qed

```

16.5 Link with NOT's DPLL

interpretation dpll_W-NOT: dpll-with-backtrack .

lemma state-eq_{NOT}-iff-eq[iff, simp]: dpll_W-NOT.state-eq_{NOT} S T \longleftrightarrow S = T
 unfolding dpll_W-NOT.state-eq_{NOT}-def by (cases S, cases T) auto

declare dpll_W-NOT.state-simp_{NOT}[simp del]

lemma dpll_W-dpll_W-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 dpllW-NOT.dpll-bj** ([], N) (M, N) and conclusive-dpllW-state (M, N)
shows M ⊨asm N ⟷ satisfiable (set-mset N)
proof –
  have dpllW-all-inv ([], N)
  unfolding dpllW-all-inv-def by auto
  show ?thesis
  apply (rule dpllW-conclusive-state-correct)
  apply (simp add: ⟨dpllW-all-inv ([], N)⟩ assms(1) rtrancpl-dpll-rtrancpl-dpllW)
  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-rev-level :: 'v literal  $\Rightarrow$  nat  $\Rightarrow$  ('v, nat, 'a) marked-lits  $\Rightarrow$  nat where
  get-rev-level - [] = 0 |
  get-rev-level L n (Marked l level # Ls) =
    (if atm-of l = atm-of L then level else get-rev-level L level Ls) |
  get-rev-level L n (Propagated l - # Ls) =
    (if atm-of l = atm-of L then n else get-rev-level L n Ls)
```

abbreviation get-level L M \equiv get-rev-level L 0 (rev M)

lemma get-rev-level-uminus[simp]: get-rev-level ($-L$) n M = get-rev-level L n M
by (induct M arbitrary: n rule: get-rev-level.induct) auto

lemma atm-of-notin-get-rev-level-eq-0[simp]:
assumes atm-of L \notin atm-of ' lits-of M
shows get-rev-level L n M = 0
using assms **apply** (induct M arbitrary: n, simp)
by (case-tac a) auto

lemma get-rev-level-ge-0-atm-of-in:
assumes get-rev-level L n M $>$ n
shows atm-of L \in atm-of ' lits-of M
using assms **apply** (induct M arbitrary: n, simp)
by (case-tac a) fastforce+

In *get-rev-level* (resp. *get-level*), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

lemma get-rev-level-skip[simp]:
assumes atm-of L \notin atm-of ' lits-of M
shows get-rev-level L n (M @ Marked K i # M') = get-rev-level L i (Marked K i # M')
using assms **apply** (induct M arbitrary: n i, simp)
by (case-tac a) auto

lemma get-rev-level-notin-end[simp]:
assumes atm-of L \notin 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

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 $\text{atm-of } L \notin \text{atm-of 'lit-of '(set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows $\text{get-level } L (M @ \text{rev } S) = \text{get-level } L M$
using *assms* **by** (*induction* *S* *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-beginning-not-marked[simp]*:
assumes $\text{atm-of } L \notin \text{atm-of 'lit-of '(set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows $\text{get-level } L (M @ S) = \text{get-level } L M$
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 $\text{atm-of } L \notin \text{atm-of 'lit-of '(set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows $\text{get-rev-level } L 0 (\text{rev } S @ \text{rev } M) = \text{get-level } L M$
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 $M :: ('a, \text{nat}, 'b) \text{ marked-lit list}$ **and** $L :: 'a \text{ literal}$
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
and $\text{atm-of } L \in \text{atm-of 'lit-of '(set } M)$
shows $\text{get-rev-level } L n M = n$

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 $M' \equiv \text{rev } M$
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{get-level } L M = 0$
proof –
have $M: M = \text{rev } M'$
unfolding *M'-def* **by** *auto*
show *?thesis*
using *assms* **unfolding** *M* **by** (*induction* *M'* *rule*: *marked-lit-list-induct*) *auto*
qed

abbreviation $M\text{Max } M \equiv \text{Max } (\text{set-mset } M)$

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

definition *get-maximum-level* :: $'a \text{ literal multiset} \Rightarrow ('a, \text{nat}, 'b) \text{ marked-lit list} \Rightarrow \text{nat}$
where
 $\text{get-maximum-level } D M = M\text{Max } (\{\#0\# \} + \text{image-mset } (\lambda L. \text{get-level } L M) D)$

lemma *get-maximum-level-ge-get-level*:
 $L \in \# D \implies \text{get-maximum-level } D M \geq \text{get-level } L M$
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-empty[simp]*:
 $\text{get-maximum-level } \{\# \} M = 0$
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-exists-lit-of-max-level*:

$D \neq \{\#\} \implies \exists L \in \# D. \text{get-level } L \ M = \text{get-maximum-level } D \ M$
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*]:
 $\text{get-maximum-level } D \ [] = 0$
unfolding *get-maximum-level-def* **by** (*simp add: image-constant-conv*)

lemma *get-maximum-level-single*[*simp*]:
 $\text{get-maximum-level } \{\#L\# \} \ M = \text{get-level } L \ M$
unfolding *get-maximum-level-def* **by** *simp*

lemma *get-maximum-level-plus*:
 $\text{get-maximum-level } (D + D') \ M = \max (\text{get-maximum-level } D \ M) (\text{get-maximum-level } D' \ M)$
by (*induct* *D*) (*auto simp add: get-maximum-level-def*)

lemma *get-maximum-level-exists-lit*:
assumes $n: n > 0$
and $\text{max: get-maximum-level } D \ M = n$
shows $\exists L \in \# D. \text{get-level } L \ M = n$
proof –
have $f: \text{finite } (\text{insert } 0 ((\lambda L. \text{get-level } L \ M) \text{ `set-mset } D))$ **by** *auto*
hence $n \in ((\lambda L. \text{get-level } L \ M) \text{ `set-mset } D)$
using $n \ \text{max} \ \text{Max-in}[OF \ f]$ **unfolding** *get-maximum-level-def* **by** *simp*
thus $\exists L \in \# D. \text{get-level } L \ M = n$ **by** *auto*
qed

lemma *get-maximum-level-skip-first*[*simp*]:
assumes $\text{atm-of } L \notin \text{atms-of } D$
shows $\text{get-maximum-level } D \ (\text{Propagated } L \ C \ \# \ M) = \text{get-maximum-level } D \ M$
using *assms* **unfolding** *get-maximum-level-def* *atms-of-def*
 $\text{atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set}$
by (*smt atm-of-in-atm-of-set-in-uminus get-level-skip-beginning image-iff marked-lit.sel(2)*
 $\text{multiset.map-cong0}$)

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
 get-maximum-possible-level [] = 0 |
 get-maximum-possible-level (Marked K i # l) = max i (get-maximum-possible-level l) |
 get-maximum-possible-level (Propagated - - # l) = get-maximum-possible-level l

lemma get-maximum-possible-level-append[simp]:
 get-maximum-possible-level (M @ M')
 = max (get-maximum-possible-level M) (get-maximum-possible-level M')
 apply (induct M, simp) by (case-tac a, auto)

lemma get-maximum-possible-level-rev[simp]:
 get-maximum-possible-level (rev M) = get-maximum-possible-level M
 apply (induct M, simp) by (case-tac a, auto)

lemma get-maximum-possible-level-ge-get-rev-level:
 max (get-maximum-possible-level M) i \geq 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 \geq 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 \geq 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* \Rightarrow '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 $xs = [] \longleftrightarrow (\forall x \in \text{set } xs. \neg \text{is-marked } x)$
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*:
 $i \in \text{set } (\text{get-all-levels-of-marked } M) \longleftrightarrow (\exists c K c'. M = c @ \text{Marked } K i \# c') \text{ (is ?A } \longleftrightarrow \text{ ?B)}$

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* $L \in \text{atm-of ' lits-of } M$
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 $M = \text{Max } (\text{insert } 0 \text{ (set (get-all-levels-of-marked } M)))$
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 $L \in \text{atm-of ' (lits-of } M) \implies \text{get-rev-level } L \ n \ M \in \{n\} \cup \text{set } (\text{get-all-levels-of-marked } M)$
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:*
 $(\forall l \in \text{set } Ls. \neg \text{is-marked } l) \longleftrightarrow \text{get-all-levels-of-marked } Ls = []$
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* $L \ 0 \ \text{rev } M$] **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* $L \notin \text{atm-of ' (lits-of } M)$
shows *get-level* $L \ (K @ M) = \text{get-rev-level } L \ (\text{last } (0 \# \text{get-all-levels-of-marked } (\text{rev } M)))$
 $(\text{rev } K)$
using *assms*
proof (*induct* M *arbitrary: K*)
case *Nil*
thus ?*case* **by** *auto*
next
case (*Cons* $a \ M$)
hence $H: \bigwedge K. \text{get-level } L \ (K @ M)$
 $= \text{get-rev-level } L \ (\text{last } (0 \# \text{get-all-levels-of-marked } (\text{rev } M))) \ (\text{rev } K)$
by *auto*
have *get-level* $L \ ((K @ [a]) @ M)$
 $= \text{get-rev-level } L \ (\text{last } (0 \# \text{get-all-levels-of-marked } (\text{rev } M))) \ (a \# \text{rev } K)$
using $H[\text{of } K @ [a]]$ **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* $L \notin \text{atm-of ' (lits-of } M)$
and *get-all-levels-of-marked* $M = \text{rev } [\text{Suc } 0..<\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))]$
shows *get-rev-level* $L \ 0 \ (\text{rev } M @ K) = \text{get-rev-level } L \ (\text{length } (\text{get-all-levels-of-marked } M)) \ K$
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.prems(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.prems by auto
  thus ?case using Cons.prems(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.prems(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.prems(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_W* =

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]:
 $\bigwedge L \text{ st. } \text{undefined-lit } (\text{trail st}) (\text{lit-of } L) \implies \text{trail } (\text{cons-trail } L \text{ st}) = L \# \text{trail st} \text{ and}$
trail-tl-trail[simp]: $\bigwedge \text{st. trail } (\text{tl-trail st}) = \text{tl } (\text{trail st}) \text{ and}$
trail-add-init-cls[simp]:
 $\bigwedge \text{st } C. \text{no-dup } (\text{trail st}) \implies \text{trail } (\text{add-init-cls } C \text{ st}) = \text{trail st} \text{ and}$
trail-add-learned-cls[simp]:
 $\bigwedge C \text{ st. no-dup } (\text{trail st}) \implies \text{trail } (\text{add-learned-cls } C \text{ st}) = \text{trail st} \text{ and}$
trail-remove-cls[simp]:
 $\bigwedge C \text{ st. trail } (\text{remove-cls } C \text{ st}) = \text{trail st} \text{ and}$
trail-update-backtrack-lvl[simp]: $\bigwedge \text{st } C. \text{trail } (\text{update-backtrack-lvl } C \text{ st}) = \text{trail st} \text{ and}$
trail-update-conflicting[simp]: $\bigwedge C \text{ st. trail } (\text{update-conflicting } C \text{ st}) = \text{trail st} \text{ and}$

init-clss-cons-trail[simp]:
 $\bigwedge M \text{ st. undefined-lit } (\text{trail st}) (\text{lit-of } M) \implies \text{init-clss } (\text{cons-trail } M \text{ st}) = \text{init-clss st} \text{ and}$
init-clss-tl-trail[simp]:
 $\bigwedge \text{st. init-clss } (\text{tl-trail st}) = \text{init-clss st} \text{ and}$
init-clss-add-init-cls[simp]:
 $\bigwedge \text{st } C. \text{no-dup } (\text{trail st}) \implies \text{init-clss } (\text{add-init-cls } C \text{ st}) = \{\#C\} + \text{init-clss st} \text{ and}$
init-clss-add-learned-cls[simp]:
 $\bigwedge C \text{ st. no-dup } (\text{trail st}) \implies \text{init-clss } (\text{add-learned-cls } C \text{ st}) = \text{init-clss st} \text{ and}$
init-clss-remove-cls[simp]:
 $\bigwedge C \text{ st. init-clss } (\text{remove-cls } C \text{ st}) = \text{remove-mset } C (\text{init-clss st}) \text{ and}$
init-clss-update-backtrack-lvl[simp]:
 $\bigwedge \text{st } C. \text{init-clss } (\text{update-backtrack-lvl } C \text{ st}) = \text{init-clss st} \text{ and}$
init-clss-update-conflicting[simp]:
 $\bigwedge C \text{ st. init-clss } (\text{update-conflicting } C \text{ st}) = \text{init-clss st} \text{ and}$

learned-clss-cons-trail[simp]:

$\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $\text{learned-clss } (cons\text{-trail } M \text{ st}) = \text{learned-clss st and}$
 $\text{learned-clss-tl-trail[simp]:}$
 $\bigwedge st. \text{learned-clss } (tl\text{-trail st}) = \text{learned-clss st and}$
 $\text{learned-clss-add-init-cl[simp]:}$
 $\bigwedge st \ C. \text{no-dup } (trail \text{ st}) \implies \text{learned-clss } (add\text{-init-cl } C \text{ st}) = \text{learned-clss st and}$
 $\text{learned-clss-add-learned-cl[simp]:}$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies \text{learned-clss } (add\text{-learned-cl } C \text{ st}) = \{\#C\# \} + \text{learned-clss st}$
 and
 $\text{learned-clss-remove-cl[simp]:}$
 $\bigwedge C \text{ st. learned-clss } (remove\text{-cl } C \text{ st}) = \text{remove-mset } C (\text{learned-clss st}) \text{ and}$
 $\text{learned-clss-update-backtrack-lvl[simp]:}$
 $\bigwedge st \ C. \text{learned-clss } (update\text{-backtrack-lvl } C \text{ st}) = \text{learned-clss st and}$
 $\text{learned-clss-update-conflicting[simp]:}$
 $\bigwedge C \text{ st. learned-clss } (update\text{-conflicting } C \text{ st}) = \text{learned-clss st and}$

$\text{backtrack-lvl-cons-trail[simp]:}$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $\text{backtrack-lvl } (cons\text{-trail } M \text{ st}) = \text{backtrack-lvl st and}$
 $\text{backtrack-lvl-tl-trail[simp]:}$
 $\bigwedge st. \text{backtrack-lvl } (tl\text{-trail st}) = \text{backtrack-lvl st and}$
 $\text{backtrack-lvl-add-init-cl[simp]:}$
 $\bigwedge st \ C. \text{no-dup } (trail \text{ st}) \implies \text{backtrack-lvl } (add\text{-init-cl } C \text{ st}) = \text{backtrack-lvl st and}$
 $\text{backtrack-lvl-add-learned-cl[simp]:}$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies \text{backtrack-lvl } (add\text{-learned-cl } C \text{ st}) = \text{backtrack-lvl st and}$
 $\text{backtrack-lvl-remove-cl[simp]:}$
 $\bigwedge C \text{ st. backtrack-lvl } (remove\text{-cl } C \text{ st}) = \text{backtrack-lvl st and}$
 $\text{backtrack-lvl-update-backtrack-lvl[simp]:}$
 $\bigwedge st \ k. \text{backtrack-lvl } (update\text{-backtrack-lvl } k \text{ st}) = k \text{ and}$
 $\text{backtrack-lvl-update-conflicting[simp]:}$
 $\bigwedge C \text{ st. backtrack-lvl } (update\text{-conflicting } C \text{ st}) = \text{backtrack-lvl st and}$

$\text{conflicting-cons-trail[simp]:}$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $\text{conflicting } (cons\text{-trail } M \text{ st}) = \text{conflicting st and}$
 $\text{conflicting-tl-trail[simp]:}$
 $\bigwedge st. \text{conflicting } (tl\text{-trail st}) = \text{conflicting st and}$
 $\text{conflicting-add-init-cl[simp]:}$
 $\bigwedge st \ C. \text{no-dup } (trail \text{ st}) \implies \text{conflicting } (add\text{-init-cl } C \text{ st}) = \text{conflicting st and}$
 $\text{conflicting-add-learned-cl[simp]:}$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies \text{conflicting } (add\text{-learned-cl } C \text{ st}) = \text{conflicting st and}$
 $\text{conflicting-remove-cl[simp]:}$
 $\bigwedge C \text{ st. conflicting } (remove\text{-cl } C \text{ st}) = \text{conflicting st and}$
 $\text{conflicting-update-backtrack-lvl[simp]:}$
 $\bigwedge st \ C. \text{conflicting } (update\text{-backtrack-lvl } C \text{ st}) = \text{conflicting st and}$
 $\text{conflicting-update-conflicting[simp]:}$
 $\bigwedge C \text{ st. conflicting } (update\text{-conflicting } C \text{ st}) = C \text{ and}$

$\text{init-state-trail[simp]: } \bigwedge N. \text{trail } (init\text{-state } N) = [] \text{ and}$
 $\text{init-state-clss[simp]: } \bigwedge N. \text{init-clss } (init\text{-state } N) = N \text{ and}$
 $\text{init-state-learned-clss[simp]: } \bigwedge N. \text{learned-clss } (init\text{-state } N) = \{\#\} \text{ and}$
 $\text{init-state-backtrack-lvl[simp]: } \bigwedge N. \text{backtrack-lvl } (init\text{-state } N) = 0 \text{ and}$
 $\text{init-state-conflicting[simp]: } \bigwedge N. \text{conflicting } (init\text{-state } N) = C\text{-True and}$

$\text{trail-restart-state[simp]: } \text{trail } (restart\text{-state } S) = [] \text{ and}$

init-clss-restart-state[simp]: *init-clss* (*restart-state* *S*) = *init-clss* *S* **and**
learned-clss-restart-state[intro]: *learned-clss* (*restart-state* *S*) $\subseteq\#$ *learned-clss* *S* **and**
backtrack-lvl-restart-state[simp]: *backtrack-lvl* (*restart-state* *S*) = 0 **and**
conflicting-restart-state[simp]: *conflicting* (*restart-state* *S*) = *C-True*
begin

definition *clauses* :: 'st \Rightarrow 'v *clauses* **where**
clauses *S* = *init-clss* *S* + *learned-clss* *S*

lemma
shows
clauses-cons-trail[simp]:
undefined-lit (*trail* *S*) (*lit-of* *M*) \Rightarrow *clauses* (*cons-trail* *M* *S*) = *clauses* *S* **and**

clss-tl-trail[simp]: *clauses* (*tl-trail* *S*) = *clauses* *S* **and**
clauses-add-learned-clss-unfolded:
no-dup (*trail* *S*) \Rightarrow *clauses* (*add-learned-clss* *U* *S*) = $\{\#U\# \} +$ *learned-clss* *S* + *init-clss* *S*
and
clauses-add-init-clss[simp]:
no-dup (*trail* *S*) \Rightarrow *clauses* (*add-init-clss* *N* *S*) = $\{\#N\# \} +$ *init-clss* *S* + *learned-clss* *S* **and**
clauses-update-backtrack-lvl[simp]: *clauses* (*update-backtrack-lvl* *k* *S*) = *clauses* *S* **and**
clauses-update-conflicting[simp]: *clauses* (*update-conflicting* *D* *S*) = *clauses* *S* **and**
clauses-remove-clss[simp]:
clauses (*remove-clss* *C* *S*) = *clauses* *S* - *replicate-mset* (*count* (*clauses* *S*) *C*) *C* **and**
clauses-add-learned-clss[simp]:
no-dup (*trail* *S*) \Rightarrow *clauses* (*add-learned-clss* *C* *S*) = $\{\#C\# \} +$ *clauses* *S* **and**
clauses-restart[simp]: *clauses* (*restart-state* *S*) $\subseteq\#$ *clauses* *S* **and**
clauses-init-state[simp]: $\bigwedge N.$ *clauses* (*init-state* *N*) = *N*
prefer 9 **using** *clauses-def* *learned-clss-restart-state* **apply** *fastforce*
by (*auto* *simp*: *ac-simps* *replicate-mset-plus* *clauses-def* *intro*: *multiset-eqI*)

abbreviation *state* :: 'st \Rightarrow ('v, nat, 'v *clause*) *marked-lit* *list* \times 'v *clauses* \times 'v *clauses*
 \times nat \times 'v *clause* *conflicting-clause* **where**
state *S* \equiv (*trail* *S*, *init-clss* *S*, *learned-clss* *S*, *backtrack-lvl* *S*, *conflicting* *S*)

abbreviation *incr-lvl* :: 'st \Rightarrow 'st **where**
incr-lvl *S* \equiv *update-backtrack-lvl* (*backtrack-lvl* *S* + 1) *S*

definition *state-eq* :: 'st \Rightarrow 'st \Rightarrow bool (*infix* \sim 50) **where**
S \sim *T* \longleftrightarrow *state* *S* = *state* *T*

lemma *state-eq-ref*[*simp*, *intro*]:
S \sim *S*
unfolding *state-eq-def* **by** *auto*

lemma *state-eq-sym*:
S \sim *T* \longleftrightarrow *T* \sim *S*
unfolding *state-eq-def* **by** *auto*

lemma *state-eq-trans*:
S \sim *T* \Rightarrow *T* \sim *U* \Rightarrow *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-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*[*intro*]:
 $x \in \text{atms-of-msu } (\text{learned-clss } (\text{restart-state } S)) \implies x \in \text{atms-of-msu } (\text{learned-clss } S)$
by (*meson atms-of-ms-mono learned-clss-restart-state set-mset-mono subsetCE*)

function *reduce-trail-to* :: 'a list \Rightarrow 'st \Rightarrow 'st **where**
reduce-trail-to $F S =$
 (if $\text{length } (\text{trail } S) = \text{length } F \vee \text{trail } S = []$ 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*]:
 $\text{trail } (\text{reduce-trail-to } [] S) = []$
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*:
 $\text{clauses } (\text{reduce-trail-to } [] S) = \text{clauses } S$
proof (*induction* $[] S$ *rule: reduce-trail-to.induct*)
case (*1 Sa*)
then have $\text{clauses } (\text{reduce-trail-to } ([::'a \text{ list}] (\text{tl-trail } Sa))) = \text{clauses } (\text{tl-trail } Sa)$
 $\vee \text{trail } Sa = []$

by fastforce
 then show clauses (reduce-trail-to ($\square::'a$ list) Sa) = clauses Sa
 by (metis (no-types) length-0-conv reduce-trail-to-eq-length clss-tl-trail
 reduce-trail-to-length-ne)
 qed

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 clss-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_{NOT}-compatible:
 assumes ST: $S \sim T$
 shows reduce-trail-to F S \sim 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 # \square])
 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 set (get-all-marked-decomposition *M*)
shows *a* = [] \vee (is-marked (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 set (get-all-marked-decomposition (trail *S*))
shows trail (reduce-trail-to *M1 S*) = *M1*
proof –
obtain *K mark* **where**
L: *L* = Marked *K 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 ([:: 'v list] 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-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-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 undefined-lit $M L \Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$
 $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L (k+1)) (\text{incr-lvl } S)$
 $\Rightarrow \text{decide } S T$

inductive-cases *decideE*[elim]: *decide* $S S'$

thm *decideE*

inductive *skip* :: 'st \Rightarrow 'st \Rightarrow bool **where**

skip-rule[intro]: state $S = (\text{Propagated } L C' \# M, N, U, k, C\text{-Clause } D) \Rightarrow -L \notin \# D \Rightarrow D \neq \{\#\}$
 $\Rightarrow T \sim \text{tl-trail } S$
 $\Rightarrow \text{skip } S T$

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 (*Propagated* L ($C + \{\#L\# \}$) $\# M$) = k

inductive *resolve* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**

resolve-rule[*intro*]:

state $S = (\text{Propagated } L (C + \{\#L\# \})) \# M, N, U, k, C\text{-Clause } (D + \{\#-L\# \})$

\Rightarrow *get-maximum-level* D (*Propagated* L ($C + \{\#L\# \}$) $\# M$) = k

$\Rightarrow T \sim \text{update-conflicting } (C\text{-Clause } (D \# \cup C)) \text{ (tl-trail } S)$

$\Rightarrow \text{resolve } S T$

inductive-cases *resolveE*[*elim*]: *resolve* $S S'$

thm *resolveE*

inductive *restart* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**

restart: *state* $S = (M, N, U, k, C\text{-True}) \Rightarrow \neg M \models_{asm} \text{clauses } S$

$\Rightarrow T \sim \text{restart-state } S$

$\Rightarrow \text{restart } S T$

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* $S = (M, N, \{\#C\# \} + U, k, C\text{-True})$

$\Rightarrow \neg M \models_{asm} \text{clauses } S$

$\Rightarrow C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$

$\Rightarrow C \notin \# \text{init-clss } S$

$\Rightarrow C \in \# \text{learned-clss } S$

$\Rightarrow T \sim \text{remove-cl } C S$

$\Rightarrow \text{forget } S T$

inductive-cases *forgetE*[*elim*]: *forget* $S T$

inductive *cdcl_W-rf* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **for** $S ::$ '*st* **where**

restart: *restart* $S T \Rightarrow \text{cdcl}_W\text{-rf } S T$ |

forget: *forget* $S T \Rightarrow \text{cdcl}_W\text{-rf } S T$

inductive *cdcl_W-bj* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**

skip[*intro*]: *skip* $S S' \Rightarrow \text{cdcl}_W\text{-bj } S S'$ |

resolve[*intro*]: *resolve* $S S' \Rightarrow \text{cdcl}_W\text{-bj } S S'$ |

backtrack[*intro*]: *backtrack* $S S' \Rightarrow \text{cdcl}_W\text{-bj } S S'$

inductive-cases *cdcl_W-bjE*: *cdcl_W-bj* $S T$

inductive *cdcl_W-o*: '*st* \Rightarrow '*st* \Rightarrow *bool* **for** $S ::$ '*st* **where**

decide[*intro*]: *decide* $S S' \Rightarrow \text{cdcl}_W\text{-o } S S'$ |

bj[*intro*]: *cdcl_W-bj* $S S' \Rightarrow \text{cdcl}_W\text{-o } S S'$

inductive *cdcl_W* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **for** $S ::$ '*st* **where**

propagate: *propagate* $S S' \Rightarrow \text{cdcl}_W S S'$ |

conflict: *conflict* $S S' \Rightarrow \text{cdcl}_W S S'$ |

other: *cdcl_W-o* $S S' \Rightarrow \text{cdcl}_W S S'$ |

rf: *cdcl_W-rf* $S S' \Rightarrow \text{cdcl}_W S S'$

lemma *rtrancp-propagate-is-rtrancp-cdcl_W*:

*propagate*** $S S' \Rightarrow \text{cdcl}_W^{**} S S'$

by (*induction rule*: *rtrancp-induct*) (*fastforce dest*!: *propagate*) +

lemma *cdcl_W-all-rules-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

```

fixes  $S :: 'st$ 
assumes
   $cdcl_W: cdcl_W\ S\ S'$  and
   $propagate: \bigwedge T. propagate\ S\ T \implies P\ S\ T$  and
   $conflict: \bigwedge T. conflict\ S\ T \implies P\ S\ T$  and
   $forget: \bigwedge T. forget\ S\ T \implies P\ S\ T$  and
   $restart: \bigwedge T. restart\ S\ T \implies P\ S\ T$  and
   $decide: \bigwedge T. decide\ S\ T \implies P\ S\ T$  and
   $skip: \bigwedge T. skip\ S\ T \implies P\ S\ T$  and
   $resolve: \bigwedge T. resolve\ S\ T \implies P\ S\ T$  and
   $backtrack: \bigwedge T. backtrack\ S\ T \implies P\ S\ T$ 
shows  $P\ S\ S'$ 
using  $assms(1)$ 
proof ( $induct\ S'$  rule:  $cdcl_W.induct$ )
  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 ( $induct\ rule: cdcl_W-o.induct$ )
      case ( $decide\ U$ )
      then show  $?case$  using  $assms(6)$  by auto
    next
      case ( $bj\ S'$ )
      then show  $?case$  using  $assms(7-9)$  by ( $induction\ rule: cdcl_W-bj.induct$ ) auto
    qed
next
  case ( $rf\ S'$ )
  then show  $?case$ 
    by ( $induct\ rule: cdcl_W-rf.induct$ ) ( $fast\ dest: forget\ restart$ ) +
qed

lemma  $cdcl_W-all-induct[consumes\ 1, case-names\ propagate\ conflict\ forget\ restart\ decide\ skip$ 
   $resolve\ backtrack]$ :
fixes  $S :: 'st$ 
assumes
   $cdcl_W: cdcl_W\ S\ S'$  and
   $propagateH: \bigwedge C\ L\ T. C + \{\#L\# \} \in \# clauses\ S \implies trail\ S \models_{as} CNot\ C$ 
     $\implies undefined-lit\ (trail\ S)\ L \implies conflicting\ S = C-True$ 
     $\implies T \sim cons-trail\ (Propagated\ L\ (C + \{\#L\# \}))\ S$ 
     $\implies P\ S\ T$  and
   $conflictH: \bigwedge D\ T. D \in \# clauses\ S \implies conflicting\ S = C-True \implies trail\ S \models_{as} CNot\ D$ 
     $\implies T \sim update-conflicting\ (C-Clause\ D)\ S$ 
     $\implies P\ S\ T$  and
   $forgetH: \bigwedge C\ T. \neg trail\ S \models_{asm} clauses\ S$ 
     $\implies C \notin set\ (get-all-mark-of-propagated\ (trail\ S))$ 
     $\implies C \notin \# init-clss\ S$ 
     $\implies C \in \# learned-clss\ S$ 
     $\implies conflicting\ S = C-True$ 
     $\implies T \sim remove-cls\ C\ S$ 
     $\implies P\ S\ T$  and
   $restartH: \bigwedge T. \neg trail\ S \models_{asm} clauses\ S$ 

```

```

     $\Rightarrow$  conflicting  $S = C\text{-True}$ 
     $\Rightarrow T \sim \text{restart-state } S$ 
     $\Rightarrow P \ S \ T$  and
  decideH:  $\bigwedge L \ T. \text{conflicting } S = C\text{-True} \Rightarrow \text{undefined-lit } (\text{trail } S) \ L$ 
     $\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L \ (\text{backtrack-lvl } S + 1)) \ (\text{incr-lvl } S)$ 
     $\Rightarrow P \ S \ T$  and
  skipH:  $\bigwedge L \ C' \ M \ D \ T. \text{trail } S = \text{Propagated } L \ C' \ \# \ M$ 
     $\Rightarrow \text{conflicting } S = C\text{-Clause } D \Rightarrow -L \notin \# \ D \Rightarrow D \neq \{\#\}$ 
     $\Rightarrow T \sim \text{tl-trail } S$ 
     $\Rightarrow P \ S \ T$  and
  resolveH:  $\bigwedge L \ C \ M \ D \ T.$ 
    trail  $S = \text{Propagated } L \ ( (C + \{\#L\#\}) \ \# \ M$ 
     $\Rightarrow \text{conflicting } S = C\text{-Clause } (D + \{\#-L\#\})$ 
     $\Rightarrow \text{get-maximum-level } D \ (\text{Propagated } L \ ( (C + \{\#L\#\}) \ \# \ M) = \text{backtrack-lvl } S$ 
     $\Rightarrow T \sim (\text{update-conflicting } (C\text{-Clause } (D \ \# \cup \ C)) \ (\text{tl-trail } S))$ 
     $\Rightarrow P \ S \ T$  and
  backtrackH:  $\bigwedge K \ i \ M1 \ M2 \ L \ D \ T.$ 
    (Marked  $K \ (\text{Suc } i) \ \# \ M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
     $\Rightarrow \text{get-level } L \ (\text{trail } S) = \text{backtrack-lvl } S$ 
     $\Rightarrow \text{conflicting } S = C\text{-Clause } (D + \{\#L\#\})$ 
     $\Rightarrow \text{get-maximum-level } (D + \{\#L\#\}) \ (\text{trail } S) = \text{get-level } L \ (\text{trail } S)$ 
     $\Rightarrow \text{get-maximum-level } D \ (\text{trail } S) \equiv i$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\#\}))$ 
      (reduce-trail-to  $M1$ 
        (add-learned-cls  $(D + \{\#L\#\})$ 
          (update-backtrack-lvl  $i$ 
            (update-conflicting  $C\text{-True } S$ ))))
     $\Rightarrow P \ S \ T$ 
  shows  $P \ S \ S'$ 
  using cdclW
  proof (induct  $S \ S'$  rule: cdclW-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_W-o-induct*[consumes 1, case-names decide skip resolve backtrack]:
fixes $S :: 'st$
assumes *cdcl_W*: *cdcl_W-o* S T **and**
 $decideH$: $\bigwedge L$ T . *conflicting* $S = C\text{-True} \implies \text{undefined-lit } (\text{trail } S) \ L$
 $\implies \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$
 $\implies T \sim \text{cons-trail } (\text{Marked } L \ (\text{backtrack-lvl } S + 1)) \ (\text{incr-lvl } S)$
 $\implies P \ S \ T$ **and**
 $skipH$: $\bigwedge L \ C' \ M \ D \ T$. *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_W* **apply** (*induct* T rule: *cdcl_W-o.induct*)
using *assms*(2) **apply** *auto*[1]
apply (*elim* *cdcl_W-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_W-o.induct*

lemma *cdcl_W-o-all-rules-induct*[consumes 1, case-names decide backtrack skip resolve]:

fixes $S \ T :: 'st$

assumes

cdcl_W-o $S \ T$ **and**

$\bigwedge T$. *decide* $S \ T \implies P \ S \ T$ **and**

$\bigwedge T$. *backtrack* $S \ T \implies P \ S \ T$ **and**

$\bigwedge T$. *skip* $S \ T \implies P \ S \ T$ **and**

$\bigwedge T$. *resolve* $S \ T \implies P \ S \ T$

shows $P \ S \ T$

using *assms* **by** (*induct* T rule: *cdcl_W-o.induct*) (*auto* *simp*: *cdcl_W-bj.simps*)

lemma *cdcl_W-o-rule-cases*[consumes 1, case-names decide backtrack skip resolve]:

fixes $S \ T :: 'st$

assumes

```

  cdclW-o S T and
  decide S T  $\implies$  P and
  backtrack S T  $\implies$  P and
  skip S T  $\implies$  P and
  resolve S T  $\implies$  P
shows P
using assms by (auto simp: cdclW-o.simps cdclW-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: get-level L (trail S) = backtrack-lvl S
  and M1: (Marked K (i + 1) # M1, M2) ∈ set (get-all-marked-decomposition (trail S))
  and no-dup: no-dup (trail S)
  and bt-l: backtrack-lvl S = length (get-all-levels-of-marked (trail S))
  and order: get-all-levels-of-marked (trail S)
    = rev ([1.. $(1 + \text{length (get-all-levels-of-marked (trail S))})$ ])
  shows atm-of L  $\notin$  atm-of ' lits-of M1

```

proof

```

  let ?M = trail S
  assume L-in-M1: atm-of L ∈ atm-of ' lits-of M1
  obtain c where Mc: trail S = c @ M2 @ Marked K (i + 1) # M1 using M1 by blast
  have atm-of L  $\notin$  atm-of ' lits-of c
    using L-in-M1 no-dup mk-disjoint-insert unfolding Mc lits-of-def by force
  have g-M-eq-g-M1: get-level L ?M = get-level L M1
    using L-in-M1 unfolding Mc by auto
  have g: get-all-levels-of-marked M1 = 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 Max (set (0 # get-all-levels-of-marked (rev M1))) < Suc i by auto
  then have get-level L M1 < Suc i
    using get-rev-level-less-max-get-all-levels-of-marked[of L 0 rev M1] by linarith
  moreover have Suc i ≤ backtrack-lvl S using bt-l by (simp add: Mc g)
  ultimately show False using L g-M-eq-g-M1 by auto

```

qed

lemma *cdcl_W-distinctinv-1*:

```

  assumes
    cdclW 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.. $(1 + \text{length (get-all-levels-of-marked (trail S))})$ ]
  shows no-dup (trail S')
  using assms

```

proof (induct rule: cdcl_W-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 $atm-of\ L \notin (\lambda l. atm-of\ (lit-of\ l))$ ‘ set $M1$
 using *backtrack-lit-skipped*[*of* $L\ S\ K\ i\ M1\ M2$] L *decomp* *backtrack.premis*
 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_W-consistent-inv-2*:

assumes
 cdcl_W $S\ S'$ 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_W-distinctinv-1*[*OF* *assms*] *distinctconsistent-interp* by *fast*

lemma *cdcl_W-o-bt*:

assumes
 cdcl_W-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_W-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 $atm-of\ L \notin (\lambda l. atm-of\ (lit-of\ l))$ ‘ 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_W-rf-bt*:

assumes
 cdcl_W-rf $S\ S'$ and
 backtrack-lvl $S = length\ (get-all-levels-of-marked\ (trail\ S))$ and
 get-all-levels-of-marked (*trail* S) = *rev* [$1..<(1+length\ (get-all-levels-of-marked\ (trail\ S)))$]
 shows *backtrack-lvl* $S' = length\ (get-all-levels-of-marked\ (trail\ S'))$
 using *assms* by (*induct rule: cdcl_W-rf.induct*) *auto*

lemma *cdcl_W-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.. $(1 + \text{length (get-all-levels-of-marked (trail S))})$ ]]) 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.. $(1 + \text{length (get-all-levels-of-marked (trail S))})$ ]]) and
  n-d: no-dup (trail S)
shows get-all-levels-of-marked (trail S')
  = rev ([1.. $(1 + \text{length (get-all-levels-of-marked (trail S'))})$ ]])
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.. $(1 + \text{length (get-all-levels-of-marked ?M)})$ ]
  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.. $(1 + \text{length (get-all-levels-of-marked ?M)})$ ]
  using H by auto
moreover have ... = rev [Suc 0.. $(1 + \text{length (get-all-levels-of-marked ?M)})$ ]
  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.. $(2 + \text{length (get-all-levels-of-marked c)} + (\text{length (get-all-levels-of-marked M2)}$ 
    + 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 $cdcl_W$ - M -level-inv ($S :: 'st$) \longleftrightarrow
 $consistent_interp \ (lits_of \ (trail \ S))$
 $\wedge no_dup \ (trail \ S)$
 $\wedge backtrack_lvl \ S = length \ (get_all_levels_of_marked \ (trail \ S))$
 $\wedge get_all_levels_of_marked \ (trail \ S)$
 $= rev \ ([1..<1+length \ (get_all_levels_of_marked \ (trail \ S))])$

lemma $cdcl_W$ - M -level-inv-decomp:
assumes $cdcl_W$ - M -level-inv S
shows $consistent_interp \ (lits_of \ (trail \ S))$
and $no_dup \ (trail \ S)$
using *assms* **unfolding** $cdcl_W$ - M -level-inv-def **by** *fastforce*+

lemma $cdcl_W$ -consistent-inv:
fixes $S \ S' :: 'st$
assumes
 $cdcl_W \ S \ S'$ **and**
 $cdcl_W$ - M -level-inv S
shows $cdcl_W$ - M -level-inv S'
using *assms* $cdcl_W$ -consistent-inv-2 $cdcl_W$ -distinctinv-1 $cdcl_W$ -bt $cdcl_W$ -bt-level'
unfolding $cdcl_W$ - M -level-inv-def **by** *meson*+

lemma $rtrancpl$ - $cdcl_W$ -consistent-inv:
assumes $cdcl_W^{**} \ S \ S'$
and $cdcl_W$ - M -level-inv S
shows $cdcl_W$ - M -level-inv S'
using *assms* **by** (induct rule: $rtrancpl$ -induct)
(auto intro: $cdcl_W$ -consistent-inv)

lemma $trancpl$ - $cdcl_W$ -consistent-inv:
assumes $cdcl_W^{++} \ S \ S'$
and $cdcl_W$ - M -level-inv S
shows $cdcl_W$ - M -level-inv S'
using *assms* **by** (induct rule: $trancpl$ -induct)
(auto intro: $cdcl_W$ -consistent-inv)

lemma $cdcl_W$ - M -level-inv-S0- $cdcl_W$ [simp]:
 $cdcl_W$ - M -level-inv (init-state N)
unfolding $cdcl_W$ - M -level-inv-def **by** *auto*

lemma $cdcl_W$ - M -level-inv-get-level-le-backtrack-lvl:
assumes *inv*: $cdcl_W$ - M -level-inv S
shows $get_level \ L \ (trail \ S) \leq backtrack_lvl \ S$
proof –
have $get_all_levels_of_marked \ (trail \ S) = rev \ [1..<1 + backtrack_lvl \ S]$
using *inv* **unfolding** $cdcl_W$ - 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: $cdcl_W$ - 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 $?M = \text{trail } S$
have
 $g: \text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [\text{Suc } 0..<\text{Suc } (\text{backtrack-lvl } S)]$
 using $M\text{-l unfolding } \text{cdcl}_W\text{-}M\text{-level-inv-def by simp-all}$
then have $i+1 \in \text{set } (\text{get-all-levels-of-marked } (\text{trail } S))$
 using $i\text{-}S$ **by** auto

then obtain $c K c'$ **where** $\text{tr-}S: \text{trail } S = c @ \text{Marked } K (i + 1) \# c'$
 using $\text{in-get-all-levels-of-marked-iff-decomp}[of i+1 \text{ trail } S]$ **by** auto

obtain $M1 M2$ **where** $(\text{Marked } K (i + 1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$
 unfolding $\text{tr-}S$ **apply** $(\text{induct } c \text{ rule: marked-lit-list-induct})$
 apply $\text{auto}[2]$
 apply $(\text{case-tac } hd (\text{get-all-marked-decomposition } (xs @ \text{Marked } K (\text{Suc } i) \# c')))$
 apply $(\text{case-tac } \text{get-all-marked-decomposition } (xs @ \text{Marked } K (\text{Suc } i) \# c'))$
 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
 $bt: \text{backtrack } S T$ **and**
 $inv: \text{cdcl}_W\text{-}M\text{-level-inv } S$ **and**
 $\text{backtrackH}: \bigwedge K i M1 M2 L D T.$
 $(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$
 $\implies \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S$
 $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$
 $\implies \text{get-level } L (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$
 $\implies \text{get-maximum-level } D (\text{trail } S) \equiv i$
 $\implies \text{undefined-lit } M1 L$
 $\implies T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (D + \{\#L\# \})$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting } C\text{-True } S))))$
 $\implies P S T$
shows $P S T$
proof –
obtain $K i M1 M2 L D$ **where**
 $\text{decomp}: (\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ **and**
 $L: \text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S$ **and**
 $\text{conf}: \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ **and**
 $\text{lev-L}: \text{get-level } L (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$ **and**
 $\text{lev-D}: \text{get-maximum-level } D (\text{trail } S) \equiv i$ **and**
 $T: T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (D + \{\#L\# \})$
 $(\text{update-backtrack-lvl } i$

```

      (update-conflicting C-True S))))
using bt by (elim backtrackE) metis

have atm-of L  $\notin$  ( $\lambda l. \text{atm-of } (\text{lit-of } l)$ ) ‘ set M1
  using backtrack-lit-skipped[of L S K i M1 M2] L decomp bt confl lev-L lev-D inv
  unfolding cdclW-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 cdclW-all-induct-lev-full:
  fixes S :: 'st
  assumes
    cdclW: cdclW S S' and
    inv[simp]: cdclW-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:  $\bigwedge C T. \neg \text{trail } S \models_{asm} \text{clauses } S$ 
       $\implies C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$ 
       $\implies C \notin \# \text{ init-clss } S$ 
       $\implies C \in \# \text{ learned-clss } S$ 
       $\implies \text{conflicting } S = C\text{-True}$ 
       $\implies T \sim \text{remove-cl } C S$ 
       $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
       $\implies P S T$  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:  $\bigwedge L T. \text{conflicting } S = C\text{-True} \implies \text{undefined-lit } (\text{trail } S) L$ 
       $\implies \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
       $\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$ 
       $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
       $\implies P S T$  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:  $\bigwedge L C M D T.$ 
       $\text{trail } S = \text{Propagated } L ( (C + \{\#L\# \})) \# M$ 
       $\implies \text{conflicting } S = C\text{-Clause } (D + \{\#-L\# \})$ 
       $\implies \text{get-maximum-level } D (\text{Propagated } L ( (C + \{\#L\# \})) \# M) = \text{backtrack-lvl } S$ 

```

```

    ⇒  $T \sim (\text{update-conflicting } (C\text{-Clause } (D \# \cup C)) \text{ (tl-trail } S))$ 
    ⇒  $\text{cdcl}_W\text{-M-level-inv } S$ 
    ⇒  $P \ S \ T$  and
backtrackH:  $\bigwedge K \ i \ M1 \ M2 \ L \ D \ T.$ 
  (Marked  $K \ (\text{Suc } i) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
  ⇒  $\text{get-level } L \ (\text{trail } S) = \text{backtrack-lvl } S$ 
  ⇒  $\text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ 
  ⇒  $\text{get-maximum-level } (D + \{\#L\# \}) \ (\text{trail } S) = \text{get-level } L \ (\text{trail } S)$ 
  ⇒  $\text{get-maximum-level } D \ (\text{trail } S) \equiv i$ 
  ⇒  $\text{undefined-lit } M1 \ L$ 
  ⇒  $T \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\# \}))$ 
    (reduce-trail-to  $M1$ 
      (add-learned-cls  $(D + \{\#L\# \})$ 
        (update-backtrack-lvl  $i$ 
          (update-conflicting  $C\text{-True } S$ ))))
  ⇒  $\text{cdcl}_W\text{-M-level-inv } S$ 
  ⇒  $P \ S \ T$ 
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;
      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  $\text{cdcl}_W\text{-all-induct-lev2} = \text{cdcl}_W\text{-all-induct-lev-full}[\text{consumes } 2, \text{ case-names propagate conflict}$ 
   $\text{forget restart decide skip resolve backtrack}]$ 

lemmas  $\text{cdcl}_W\text{-all-induct-lev} = \text{cdcl}_W\text{-all-induct-lev-full}[\text{consumes } 1, \text{ case-names lev-inv propagate}$ 
   $\text{conflict forget restart decide skip resolve backtrack}]$ 

```

```

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:  $\bigwedge L T. \text{conflicting } S = C\text{-True} \implies \text{undefined-lit } (\text{trail } S) L$ 
       $\implies \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
       $\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$ 
       $\implies \text{cdcl}_W\text{-M-level-inv } S$ 
       $\implies P S T$  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 *cdcl_W-o-induct-lev2* = *cdcl_W-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: cdcl_W-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 cdcl_W-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_W-state-eq-compatible:

assumes
 cdcl_W S T **and** \neg restart S T **and**
 $S \sim S'$ **and**
 $T \sim T'$ **and**
 inv: cdcl_W-M-level-inv S
shows cdcl_W S' T'
using assms **by** (meson assms backtrack-state-eq-compatible bj cdcl_W.simps cdcl_W-bj.simps
 cdcl_W-o-rule-cases cdcl_W-rf.cases cdcl_W-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)

lemma cdcl_W-bj-state-eq-compatible:

assumes
 cdcl_W-bj S T **and** cdcl_W-M-level-inv S
 $S \sim S'$ **and**
 $T \sim T'$
shows cdcl_W-bj S' T'
using assms
by induction (auto
 intro: skip-state-eq-compatible backtrack-state-eq-compatible resolve-state-eq-compatible)

lemma tranclp-cdcl_W-bj-state-eq-compatible:

assumes
 cdcl_W-bj⁺⁺ S T **and** inv: cdcl_W-M-level-inv S **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows cdcl_W-bj⁺⁺ S' T'

```

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]  $\langle cdcl_W\text{-bj } T \ U \rangle$  by auto
  then show ?case
    using IH[of T] by auto
qed

```

17.4.4 Conservation of some Properties

lemma *level-of-marked-ge-1*:

```

assumes
  cdclW S S' and
  inv: cdclW-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: cdclW-all-induct-lev2)
by (auto dest: union-in-get-all-marked-decomposition-is-subset simp: cdclW-M-level-inv-decomp)

```

lemma *cdcl_W-o-no-more-init-clss*:

```

assumes
  cdclW-o S S' and
  inv: cdclW-M-level-inv S
shows init-clss S = init-clss S'
using assms by (induct rule: cdclW-o-induct-lev2) (auto simp: cdclW-M-level-inv-decomp)

```

lemma *trancpl-cdcl_W-o-no-more-init-clss*:

```

assumes
  cdclW-o++ S S' and
  inv: cdclW-M-level-inv S
shows init-clss S = init-clss S'
using assms apply (induct rule: trancpl.induct)
by (auto dest: cdclW-o-no-more-init-clss
  dest!: trancpl-cdclW-consistent-inv dest: trancpl-mono-explicit[of cdclW-o - - cdclW]
  simp: other)

```

lemma *rtrancpl-cdcl_W-o-no-more-init-clss*:

```

assumes
  cdclW-o** S S' and
  inv: cdclW-M-level-inv S
shows init-clss S = init-clss S'
using assms unfolding rtrancpl-unfold by (auto intro: trancpl-cdclW-o-no-more-init-clss)

```

lemma *cdcl_W-init-clss*:

```

cdclW S T  $\implies$  cdclW-M-level-inv S  $\implies$  init-clss S = init-clss T
by (induct rule: cdclW-all-induct-lev2) (auto simp: cdclW-M-level-inv-def)

```

lemma *rtrancpl-cdcl_W-init-clss*:

*cdcl_W** S T \implies cdcl_W-M-level-inv S \implies init-clss S = init-clss T*

by (induct rule: *rtrancpl-induct*) (auto dest: *cdcl_W-init-clss rtrancpl-cdcl_W-consistent-inv*)

lemma *trancpl-cdcl_W-init-clss*:

*cdcl_W** S T \implies cdcl_W-M-level-inv S \implies init-clss S = init-clss T*

using *rtrancpl-cdcl_W-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_W-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 *cdcl_W-learned-clause-S0-cdcl_W[simp]*:

cdcl_W-learned-clause (init-state N)

unfolding *cdcl_W-learned-clause-def* **by** *auto*

lemma *cdcl_W-learned-clss*:

assumes

cdcl_W S S' and

learned: cdcl_W-learned-clause S and

lev-inv: cdcl_W-M-level-inv S

shows *cdcl_W-learned-clause S'*

using *assms(1) lev-inv learned*

proof (induct rule: *cdcl_W-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 cdcl_W-learned-clause-def*

by (auto dest!: *get-all-marked-decomposition-exists-prepend*

simp: clauses-def cdcl_W-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 \models_{psm} learned-clss S*

using *learned trail unfolding cdcl_W-learned-clause-def clauses-def* **by** *auto*

then have *init-clss S \models_{pm} C + {#L#}*

using *trail learned unfolding cdcl_W-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
    cdclW** S S' and
    cdclW-M-level-inv S
    cdclW-learned-clause S
  shows cdclW-learned-clause S'
  using assms by induction (auto dest: cdclW-learned-clss intro: rtrancpl-cdclW-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-strange-atm* $S' \longleftrightarrow$ (
 $(\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S'))$
 $\wedge (\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S'))$
 $\wedge \text{atms-of-msu } (\text{learned-clss } S') \subseteq \text{atms-of-msu } (\text{init-clss } S')$
 $\wedge \text{atm-of } ' (\text{lits-of } (\text{trail } S')) \subseteq \text{atms-of-msu } (\text{init-clss } S'))$

lemma *no-strange-atm-decomp*:
 assumes *no-strange-atm* S
 shows *conflicting* S = C-Clause T \implies *atms-of* T \subseteq *atms-of-msu* (*init-clss* S)
 and $(\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S))$
 and *atms-of-msu* (*learned-clss* S) \subseteq *atms-of-msu* (*init-clss* S)
 and *atm-of* ' (*lits-of* (*trail* S)) \subseteq *atms-of-msu* (*init-clss* S)
 using assms **unfolding** *no-strange-atm-def* **by** blast+

lemma *no-strange-atm-S0* [*simp*]: *no-strange-atm* (*init-state* N)
unfolding *no-strange-atm-def* **by** auto

lemma *cdcl_W-no-strange-atm-explicit*:
 assumes

$cdcl_W \ S \ S'$ and
 $lev: cdcl_W\text{-}M\text{-level-inv} \ S$ and
 $conf: \forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S)$ and
 $marked: \forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S)$
 $\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-msu } (\text{init-clss } S)$ and
 $learned: \text{atms-of-msu } (\text{learned-clss } S) \subseteq \text{atms-of-msu } (\text{init-clss } S)$ and
 $trail: \text{atm-of } ' (\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu } (\text{init-clss } S)$
shows $(\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S')) \wedge$
 $(\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S'))$
 $\longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S') \wedge$
 $\text{atms-of-msu } (\text{learned-clss } S') \subseteq \text{atms-of-msu } (\text{init-clss } S') \wedge$
 $\text{atm-of } ' (\text{lits-of } (\text{trail } S')) \subseteq \text{atms-of-msu } (\text{init-clss } S') \text{ (is } ?C \ S' \wedge ?M \ S' \wedge ?U \ S' \wedge ?V \ S')$
using $assms(1,2)$
proof ($induct \text{ rule: } cdcl_W\text{-all-induct-lev2}$)
case ($propagate \ C \ L \ T$) **note** $C\text{-L} = \text{this}(1)$ and $undef = \text{this}(3)$ and $confl = \text{this}(4)$ and $T = \text{this}(5)$
have $?C \ (\text{cons-trail } (\text{Propagated } L \ (C + \{\#L\# \})) \ S)$ **using** $confl \ undef$ **by** $auto$
moreover
have $\text{atms-of } (C + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{init-clss } S)$
by ($metis \ (no\text{-types}) \ \text{atms-of-atms-of-ms-mono} \ \text{atms-of-ms-union} \ \text{clauses-def} \ \text{mem-set-mset-iff}$
 $C\text{-L} \ \text{learned} \ \text{set-mset-union} \ \text{sup.orderE}$)
then have $?M \ (\text{cons-trail } (\text{Propagated } L \ (C + \{\#L\# \})) \ S)$ **using** $undef$
by ($simp \ add: \text{marked}$)
moreover have $?U \ (\text{cons-trail } (\text{Propagated } L \ (C + \{\#L\# \})) \ S)$
using $learned \ undef$ **by** $auto$
moreover have $?V \ (\text{cons-trail } (\text{Propagated } L \ (C + \{\#L\# \})) \ S)$
using $C\text{-L} \ \text{learned} \ \text{trail} \ \text{undef} \ \text{unfolding} \ \text{clauses-def}$
by ($auto \ simp: \text{in-plus-implies-atm-of-on-atms-of-ms}$)
ultimately show $?case$ **using** T **by** $auto$
next
case ($decide \ L$)
then show $?case$ **using** $learned \ marked \ confl \ trail$ **unfolding** $clauses-def$ **by** $auto$
next
case ($skip \ L \ C \ M \ D$)
then show $?case$ **using** $learned \ marked \ confl \ trail$ **by** $auto$
next
case ($conflict \ D \ T$) **note** $T = \text{this}(4)$
have $D: \text{atm-of } ' \ \text{set-mset } D \subseteq \bigcup (\text{atms-of } ' \ (\text{set-mset } (\text{clauses } S)))$
using $\langle D \in \# \ \text{clauses } S \rangle$ **by** ($auto \ simp \ add: \text{atms-of-def} \ \text{atms-of-ms-def}$)
moreover {
fix $xa :: 'v \ \text{literal}$
assume $a1: \text{atm-of } ' \ \text{set-mset } D \subseteq (\bigcup x \in \text{set-mset } (\text{init-clss } S). \text{atms-of } x)$
 $\cup (\bigcup x \in \text{set-mset } (\text{learned-clss } S). \text{atms-of } x)$
assume $a2: (\bigcup x \in \text{set-mset } (\text{learned-clss } S). \text{atms-of } x) \subseteq (\bigcup x \in \text{set-mset } (\text{init-clss } S). \text{atms-of } x)$
assume $xa \in \# \ D$
then have $\text{atm-of } xa \in \text{UNION } (\text{set-mset } (\text{init-clss } S)) \ \text{atms-of}$
using $a2 \ a1$ **by** ($metis \ (no\text{-types}) \ \text{Un-iff} \ \text{atm-of-lit-in-atms-of} \ \text{atms-of-def} \ \text{subset-Un-eq}$)
then have $\exists m \in \text{set-mset } (\text{init-clss } S). \text{atm-of } xa \in \text{atms-of } m$
by $blast$
} **note** $H = \text{this}$
ultimately show $?case$ **using** $conflict.premis \ T \ \text{learned} \ \text{marked} \ \text{confl} \ \text{trail}$
unfolding $\text{atms-of-def} \ \text{atms-of-ms-def} \ \text{clauses-def}$
by ($auto \ simp \ add: \ H$)
next
case ($restart \ T$)
then show $?case$ **using** $learned \ marked \ confl \ trail$ **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:  $\bigwedge L$  mark. Propagated L mark  $\in$  set (trail S)  $\implies$  atms-of mark  $\subseteq$  atms-of-msu (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-ms-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  $\subseteq$  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: cdclW-M-level-inv-decomp)
moreover have ?V T
  using M conf confl trail T undef decomp lev by (force simp: cdclW-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 = update-conflicting (C-Clause (remdups-mset (D + C))) (tl-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 cdclW-no-strange-atm-inv:
  assumes cdclW S S' and no-strange-atm S and cdclW-M-level-inv S
  shows no-strange-atm S'
  using cdclW-no-strange-atm-explicit[OF assms(1)] assms(2,3) unfolding no-strange-atm-def by fast

lemma rtranclp-cdclW-no-strange-atm-inv:
  assumes cdclW** S S' and no-strange-atm S and cdclW-M-level-inv S
  shows no-strange-atm S'
  using assms by induction (auto intro: cdclW-no-strange-atm-inv rtranclp-cdclW-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_W-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_W-state-decomp*:
assumes *distinct-cdcl_W-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_W-state-def* **by** *blast+*

lemma *distinct-cdcl_W-state-decomp-2*:
assumes *distinct-cdcl_W-state* ($S::st$)
shows $\text{conflicting } S = C\text{-Clause } T \implies \text{distinct-mset } T$
using *assms* **unfolding** *distinct-cdcl_W-state-def* **by** *auto*

lemma *distinct-cdcl_W-state-S0-cdcl_W[simp]*:
 $\text{distinct-mset-mset } N \implies \text{distinct-cdcl}_W\text{-state } (\text{init-state } N)$
unfolding *distinct-cdcl_W-state-def* **by** *auto*

lemma *distinct-cdcl_W-state-inv*:
assumes
 $\text{cdcl}_W \text{ } S \text{ } S'$ **and**
 $\text{cdcl}_W\text{-M-level-inv } S$ **and**
 $\text{distinct-cdcl}_W\text{-state } S$
shows $\text{distinct-cdcl}_W\text{-state } S'$
using *assms*
proof (*induct rule: cdcl_W-all-induct-lev2*)
case (*backtrack* $K \ i \ M1 \ M2 \ L \ D$)
then show *?case*
unfolding *distinct-cdcl_W-state-def*
by (*fastforce dest: get-all-marked-decomposition-incl simp: cdcl_W-M-level-inv-decomp*)
next
case *restart*
then show *?case* **unfolding** *distinct-cdcl_W-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-cdcl_W-state-def distinct-mset-set-def clauses-def*
 $\text{distinct-mset-single-add}$
 $\text{intro!}: \text{distinct-mset-union-mset}$)
qed (*auto simp add: distinct-cdcl_W-state-def distinct-mset-set-def clauses-def*)

lemma *rtanclp-distinct-cdcl_W-state-inv*:
assumes
 $\text{cdcl}_W^{**} \text{ } S \text{ } S'$ **and**
 $\text{cdcl}_W\text{-M-level-inv } S$ **and**

distinct-cdcl_W-state S
shows *distinct-cdcl_W-state S'*
using *assms apply (induct rule: rtrancpl-induct)*
using *distinct-cdcl_W-state-inv rtrancpl-cdcl_W-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
 $\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$
 $\longrightarrow (b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$

definition *cdcl_W-conflicting S \equiv*
 $(\forall T. \text{conflicting } S = \text{C-Clause } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T)$
 $\wedge \text{every-mark-is-a-conflict } S$

lemma *backtrack-atms-of-D-in-M1:*

fixes *M1 :: ('v, nat, 'v clause) marked-lits*
assumes
inv: cdcl_W-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: backtrack-lvl S = get-maximum-level (D + {#L#}) (trail S) and
S-conf: conflicting S = C-Clause (D + {#L#}) and
undef: undefined-lit M1 L and
T: T \sim (cons-trail (Propagated L (D + {#L#})))
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (D + \{\#L\}))$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting } \text{C-True } S))))$ **and**
conf: $\forall T. \text{conflicting } S = \text{C-Clause } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$
shows *atms-of D \subseteq atm-of ' lits-of (tl (trail T))*

proof (rule ccontr)

let *?k = get-maximum-level (D + {#L#}) (trail S)*
have *trail S \models_{as} CNot D* **using** *conf S-conf* **by** *auto*
then have *vars-of-D: atms-of D \subseteq atm-of ' lits-of (trail S)* **unfolding** *atms-of-def*
by *(meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)*

obtain *M0* **where** *M: trail S = M0 @ M2 @ Marked K (Suc i) # M1*
using *decomp* **by** *auto*

have *max: get-maximum-level (D + {#L#}) (trail S)*
 $= \text{length } (\text{get-all-levels-of-marked } (M0 @ M2 @ \text{Marked } K (Suc i) \# M1))$
using *inv unfolding cdcl_W-M-level-inv-def S-lvl M* **by** *simp*

assume *a: \neg ?thesis*

then obtain *L'* **where**

L': L' \in atms-of D and

L'-notin-M1: L' \notin atm-of ' lits-of M1

using *T undef decomp inv* **by** *(auto simp: cdcl_W-M-level-inv-decomp)*

then have *L'-in: L' \in atm-of ' lits-of (M0 @ M2 @ 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 $L' L'$ -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' -notin- $M1$ $L'' M$ by $(\text{auto simp del: get-rev-level.simps})$
have $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [1..<1+?k]$
using $\text{inv } S\text{-lvl}$ unfolding $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ 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 $(\text{auto simp: rev-swap[symmetric] dest!: append-cons-eq-upt-length-i-end})$

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 $\text{get-rev-level-ge-min-get-all-levels-of-marked[of } L''$
 $\text{rev } (M0 @ M2 @ [\text{Marked } K (\text{Suc } i)]) \text{ Suc } i] L'\text{-in}$
unfolding L'' by $(\text{fastforce simp add: lits-of-def})$
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 $(\text{simp add: Un-commute})$
also have $\dots = \text{Min } (\text{set } ((\text{Suc } i) \# [\text{Suc } (\text{Suc } i)..<2 + \text{length } (\text{get-all-levels-of-marked } M0)$
 $+ (\text{length } (\text{get-all-levels-of-marked } M2) + \text{length } (\text{get-all-levels-of-marked } M1)))]))$
unfolding M by $(\text{auto simp add: Un-commute})$
also have $\dots = \text{Suc } i$ by $(\text{auto intro: Min-eqI})$
finally have $\text{get-rev-level } L'' (\text{Suc } i) (\text{Marked } K (\text{Suc } i) \# \text{rev } (M0 @ M2)) \geq \text{Suc } i$.
then have $\text{get-level } L'' (\text{trail } S) \geq i + 1$
using $\langle \text{get-level } L'' (\text{trail } S) = \text{get-rev-level } L'' (\text{Suc } i) (\text{Marked } K (\text{Suc } i) \# \text{rev } M2 @ \text{rev } M0) \rangle$
by simp
then have $\text{get-maximum-level } D (\text{trail } S) \geq i + 1$
using $\text{get-maximum-level-ge-get-level[OF } \langle L'' \in \# D \rangle, \text{ of trail } S]$ by auto
then show False using i by auto
qed

lemma $\text{distinct-atms-of-incl-not-in-other}$:
assumes $a1: \text{no-dup } (M @ M')$
and $a2: \text{atms-of } D \subseteq \text{atm-of } ' \text{lits-of } M'$
shows $\forall x \in \text{atms-of } D. x \notin \text{atm-of } ' \text{lits-of } M$

proof –

{ fix $aa :: 'a$
have $\text{ff1}: \bigwedge l \text{ ms. undefined-lit } ms \ l \vee \text{atm-of } l$
 $\in \text{set } (\text{map } (\lambda m. \text{atm-of } (\text{lit-of } (m::('a, 'b, 'c) \text{marked-lit}))) \text{ ms})$
by $(\text{simp add: defined-lit-map})$
have $\text{ff2}: \bigwedge a. a \notin \text{atms-of } D \vee a \in \text{atm-of } ' \text{lits-of } M'$
using $a2$ by (meson subsetCE)
have $\text{ff3}: \bigwedge a. a \notin \text{set } (\text{map } (\lambda m. \text{atm-of } (\text{lit-of } m)) M')$
 $\vee a \notin \text{set } (\text{map } (\lambda m. \text{atm-of } (\text{lit-of } m)) M)$
using $a1$ by $(\text{metis (lifting) IntI distinct-append empty-iff map-append})$
have $\forall L \text{ a f. } \exists l. ((a::'a) \notin f' L \vee (l::'a \text{ literal}) \in L) \wedge (a \notin f' L \vee f l = a)$
by blast
then have $aa \notin \text{atms-of } D \vee aa \notin \text{atm-of } ' \text{lits-of } M$

```

    using ff3 ff2 ff1 by (metis (no-types) Marked-Propagated-in-iff-in-lits-of) }
  then show ?thesis
    by blast
qed

lemma cdclW-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:  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot\ T$  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)  $\models_{ps}$  (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 (λa. {#lit-of a#}) ‘ set a ∪ set-mset (init-clss S) |=ps (λa. {#lit-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 ∀ l ∈ set M2. ¬is-marked l
  using get-all-marked-decomposition-snd-not-marked backtrack.hyps(1) by blast
  obtain M0 where M: trail S = M0 @ M2 @ Marked K (i + 1) # M1
  using decomp' by auto
  show ?case unfolding all-decomposition-implies-def
  proof
    fix x
    assume x ∈ set (get-all-marked-decomposition (trail T))
    then have x: x ∈ set (get-all-marked-decomposition (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 ∨ x ∈ set ?tl
    using x by (case-tac ?m) auto
  moreover {
    assume x ∈ set ?tl
    then have x ∈ set (get-all-marked-decomposition (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) ⇒ (λa. {#lit-of a#}) ‘ set Ls
    ∪ set-mset (init-clss (T))
  }

```



```

     $\models_{ps} (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set seen}$ 
    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  $M1: hd\ (get\text{-}all\text{-}marked\text{-}decomposition\ M1) = (M1', M1'')$ 
      by (cases  $hd\ (get\text{-}all\text{-}marked\text{-}decomposition\ M1)$ )
    then have  $x': x = (M1', Propagated\ L\ (D + \{\#L\# \})) \# M1''$ 
      using  $\langle x = ?hd \rangle$  by auto
    have  $(M1', M1'') \in set\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S))$ 
      using  $M1[symmetric]\ hd\text{-}get\text{-}all\text{-}marked\text{-}decomposition\text{-}skip\text{-}some[OF\ M1[symmetric],$ 
        of  $M0 @ M2 - i + 1]$  unfolding M by fastforce
    then have  $1: (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } M1' \cup set\text{-}mset\ (init\text{-}clss\ S)$ 
       $\models_{ps} (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } M1''$ 
      using decomp unfolding all-decomposition-implies-def by auto
    moreover
      have  $trail\ S \models_{as}\ CNot\ D$  using conf confl by auto
      then have  $vars\text{-}of\text{-}D: atms\text{-}of\ D \subseteq atm\text{-}of\ \text{' lits-of } (trail\ S)$ 
        unfolding atms-of-def
        by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)
      have  $vars\text{-}of\text{-}D: atms\text{-}of\ D \subseteq atm\text{-}of\ \text{' lits-of } M1$ 
        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:
         $\forall x \in atms\text{-}of\ D. x \notin atm\text{-}of\ \text{' lits-of } (M0 @ M2 @ Marked\ K\ (i + 1) \# \square)$ 
        using vars-of-D distinct-atms-of-incl-not-in-other[of M0 @ M2 @ Marked K (i + 1) # □
          M1]
        unfolding M by auto
      have  $M1 \models_{as}\ CNot\ D$ 
        using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # □
          M1 CNot D]  $\langle trail\ S \models_{as}\ CNot\ D \rangle$  unfolding M lits-of-def by simp
      have  $M1 = M1'' @ M1'$  by (simp add: M1 get-all-marked-decomposition-decomp)
      have  $TT: (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } M1' \cup set\text{-}mset\ (init\text{-}clss\ S) \models_{ps}\ CNot\ D$ 
        using true-annots-true-clss-clc[OF ⟨M1  $\models_{as}\ CNot\ D$ ⟩ true-clss-clss-left-right[OF 1,
          of CNot D] unfolding  $(M1 = M1'' @ M1')$  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  $T': (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } M1' \cup set\text{-}mset\ (init\text{-}clss\ S) \models_p\ D + \{\#L\# \}$  by auto
      have  $atms\text{-}of\ (D + \{\#L\# \}) \subseteq atms\text{-}of\text{-}msu\ (clauses\ S)$ 
        using alien conf unfolding no-strange-atm-def clauses-def by auto
      then have  $(\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } M1' \cup set\text{-}mset\ (init\text{-}clss\ S) \models_p\ \{\#L\# \}$ 
        using true-clss-clc-plus-CNot[OF T' TT] by auto
    ultimately
      have case  $x\ of\ (Ls, seen) \Rightarrow (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } Ls$ 
         $\cup set\text{-}mset\ (init\text{-}clss\ T)$ 
         $\models_{ps} (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set seen}$  using  $T'\ T\ decomp'\ undef\ inv$  unfolding  $x'$ 
        by (simp add: cdclW-M-level-inv-decomp)
  }
  ultimately show case  $x\ of\ (Ls, seen) \Rightarrow (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set } Ls \cup set\text{-}mset\ (init\text{-}clss\ T)$ 
     $\models_{ps} (\lambda a. \{\#lit\text{-}of\ a\# \}) \text{ ' set seen}$  using T by auto
qed
qed

```

```

lemma cdclW-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-cls S) (get-all-marked-decomposition (trail S)) and
    confl:  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} C\text{Not } T$  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' mark a b
      assume a @ Propagated L' mark # b = trail T
      then have (a = []  $\wedge$  L = L'  $\wedge$  mark = C + {#L#}  $\wedge$  b = trail S)
         $\vee$  tl a @ Propagated L' mark # b = trail S
        using T undef by (cases a) fastforce+
      moreover {
        assume tl a @ Propagated L' mark # b = trail S
        then have b  $\models_{as}$  CNot ( mark - {#L'#})  $\wedge$  L'  $\in$  # mark
          using mark-confl by auto
      }
      moreover {
        assume a = [] and L = L' and mark = C + {#L#} and b = trail S
        then have b  $\models_{as}$  CNot ( mark - {#L'#})  $\wedge$  L  $\in$  # mark
          using (trail S  $\models_{as}$  CNot C) 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  $\bigwedge a \text{ La mark } b. a @ \text{Propagated La mark \# } b = \text{Marked L (backtrack-lvl S+1) \# trail S}$ 
       $\implies$  tl a @ Propagated La mark # b = 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 = 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  $\forall \text{La mark } a \text{ b. } a @ \text{Propagated La mark \# } b = \text{Propagated L C' \# M}$ 
           $\longrightarrow$  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 = 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: trail S = M0 @ M2 @ Marked K (i + 1) # M1
  using backtrack.hyps(1) by auto
have [simp]: trail (reduce-trail-to M1 (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 = []  $\wedge$  Propagated La mark = Propagated L (D + {#L#})  $\wedge$  b = M1)
     $\vee$  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  $\models_{as}$  CNot D using conf confl by auto
  then have vars-of-D: atms-of D  $\subseteq$  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  $\subseteq$  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:  $\forall x \in \text{atms-of } D. x \notin$ 
    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  $\models_{as}$  CNot D
    using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # [] M1
      CNot D] (trail S  $\models_{as}$  CNot D) unfolding M lits-of-def by simp
  then have b  $\models_{as}$  CNot ( mark - {#La#})  $\wedge$  La  $\in$  # 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  $b \models_{as} CNot\ (mark - \{\#La\}) \wedge La \in \# \ mark$ 
  using mark-conf by blast
}
ultimately show  $b \models_{as} CNot\ (mark - \{\#La\}) \wedge La \in \# \ mark$  by fast
qed
qed

```

lemma *cdcl_W-conflicting-is-false*:

```

assumes
  cdclW  $S\ S'$  and
  M-lev: cdclW-M-level-inv  $S$  and
  confl-inv:  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow trail\ S \models_{as} CNot\ T$  and
  marked-conf:  $\forall L\ mark\ a\ b. a \ @\ Propagated\ L\ mark\ \# \ b = (trail\ S) \longrightarrow (b \models_{as} CNot\ (mark - \{\#L\}) \wedge L \in \# \ mark)$  and
  dist: distinct-cdclW-state  $S$ 
shows  $\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow trail\ S' \models_{as} CNot\ T$ 
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
case (skip  $L\ C'\ M\ D$ ) note  $tr\text{-}S = this(1)$  and  $T = this(5)$ 
then have  $Propagated\ L\ C' \ \# \ M \models_{as} CNot\ D$  using assms skip by auto
moreover
  have  $L \notin \# \ D$ 
  proof (rule ccontr)
    assume  $\neg \text{?thesis}$ 
    then have  $-L \in lits\text{-of } M$ 
    using in-CNot-implies-uminus(2)[of D L Propagated L C' \# M]
     $\langle Propagated\ L\ C' \ \# \ M \models_{as} CNot\ D \rangle$  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  $tr = this(1)$  and  $confl = this(2)$  and  $T = this(4)$ 
show ?case
proof (intro allI impI)
  fix  $T'$ 
  have  $tl\ (trail\ S) \models_{as} CNot\ C$  using  $tr\ assms(4)$  by fastforce
  moreover
    have distinct-mset  $(D + \{\#-L\})$  using confl dist
    unfolding distinct-cdclW-state-def by auto
    then have  $-L \notin \# \ D$  unfolding distinct-mset-def by auto
    have  $M \models_{as} CNot\ D$ 
    proof -
      have  $Propagated\ L\ ((C + \{\#L\})) \ \# \ M \models_{as} CNot\ D \cup CNot\ \{\#-L\}$ 
      using confl tr confl-inv by force
      then show ?thesis
      using M-lev  $\langle -L \notin \# \ D \rangle$   $tr\ true\text{-annots-lit-of-notin-skip}$ 
      unfolding cdclW-M-level-inv-def by force
    qed
  qed

```

```

      qed
    moreover assume conflicting  $T = C\text{-Clause } T'$ 
    ultimately
      show trail  $T \models_{as} CNot\ T'$ 
      using tr  $T$  by auto
    qed
  qed (auto simp: assms(2) cdclW-M-level-inv-decomp)

```

```

lemma cdclW-conflicting-decomp:
  assumes cdclW-conflicting  $S$ 
  shows  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot\ T$ 
  and  $\forall L \text{ mark } a \ b. a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$ 
     $\longrightarrow (b \models_{as} CNot\ (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$ 
  using assms unfolding cdclW-conflicting-def by blast+

```

```

lemma cdclW-conflicting-decomp2:
  assumes cdclW-conflicting  $S$  and conflicting  $S = C\text{-Clause } T$ 
  shows trail  $S \models_{as} CNot\ T$ 
  using assms unfolding cdclW-conflicting-def by blast+

```

```

lemma cdclW-conflicting-decomp2':
  assumes
    cdclW-conflicting  $S$  and
    conflicting  $S = C\text{-Clause } D$ 
  shows trail  $S \models_{as} CNot\ D$ 
  using assms unfolding cdclW-conflicting-def by auto

```

```

lemma cdclW-conflicting-S0-cdclW[simp]:
  cdclW-conflicting (init-state  $N$ )
  unfolding cdclW-conflicting-def by auto

```

17.4.9 Putting all the invariants together

```

lemma cdclW-all-inv:
  assumes cdclW: cdclW  $S\ S'$  and
    1: all-decomposition-implies-m (init-clss  $S$ ) (get-all-marked-decomposition (trail  $S$ )) and
    2: cdclW-learned-clause  $S$  and
    4: cdclW-M-level-inv  $S$  and
    5: no-strange-atm  $S$  and
    7: distinct-cdclW-state  $S$  and
    8: cdclW-conflicting  $S$ 
  shows all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ ))
  and cdclW-learned-clause  $S'$ 
  and cdclW-M-level-inv  $S'$ 
  and no-strange-atm  $S'$ 
  and distinct-cdclW-state  $S'$ 
  and cdclW-conflicting  $S'$ 
proof -
  show S1: all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ ))
    using cdclW-propagate-is-conclusion[OF cdclW 4 1 2 - 5] 8 unfolding cdclW-conflicting-def
    by blast
  show S2: cdclW-learned-clause  $S'$  using cdclW-learned-clss[OF cdclW 2 4] .
  show S4: cdclW-M-level-inv  $S'$  using cdclW-consistent-inv[OF cdclW 4] .
  show S5: no-strange-atm  $S'$  using cdclW-no-strange-atm-inv[OF cdclW 5 4] .
  show S7: distinct-cdclW-state  $S'$  using distinct-cdclW-state-inv[OF cdclW 4 7] .
  show S8: cdclW-conflicting  $S'$ 

```

```

using cdclW-conflicting-is-false[OF cdclW 4 - - 7] 8 cdclW-propagate-is-false[OF cdclW 4 2 1 -
5]
unfolding cdclW-conflicting-def by fast
qed

```

lemma *rtranclp-cdcl_W-all-inv*:

assumes

cdcl_W: rtranclp cdcl_W S S' **and**

1: *all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))* **and**

2: *cdcl_W-learned-clause S* **and**

4: *cdcl_W-M-level-inv S* **and**

5: *no-strange-atm S* **and**

7: *distinct-cdcl_W-state S* **and**

8: *cdcl_W-conflicting S*

shows

all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S')) **and**

cdcl_W-learned-clause S' **and**

cdcl_W-M-level-inv S' **and**

no-strange-atm S' **and**

distinct-cdcl_W-state S' **and**

cdcl_W-conflicting S'

using *assms*

proof (*induct rule: rtranclp-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** *cdcl_W-all-inv[OF H(2)]*

H **by** *presburger*

case 2 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-all-inv[OF H(2)]*

H **by** *presburger*

case 3 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-all-inv[OF H(2)]*

H **by** *presburger*

case 4 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-all-inv[OF H(2)]*

H **by** *presburger*

case 5 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-all-inv[OF H(2)]*

H **by** *presburger*

case 6 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-all-inv[OF H(2)]*

H **by** *presburger*

qed

lemma *all-invariant-S0-cdcl_W*:

assumes *distinct-mset-mset N*

shows *all-decomposition-implies-m (init-clss (init-state N))*

(*get-all-marked-decomposition (trail (init-state N))*)

and *cdcl_W-learned-clause (init-state N)*

and $\forall T. \text{conflicting } (init-state N) = C\text{-Clause } T \longrightarrow (trail (init-state N)) \models_{as} CNot T$

and *no-strange-atm (init-state N)*

and *consistent-interp (lits-of (trail (init-state N)))*

and $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = \text{trail } (init-state N) \longrightarrow$

$(b \models_{as} CNot (mark - \{\#L\}) \wedge L \in \# mark)$
and *distinct-cdcl_W-state* (*init-state* N)
using *assms* **by** *auto*

lemma *cdcl_W-only-propagated-vars-unsat*:

assumes

marked: $\forall x \in set\ M. \neg is_marked\ x$ **and**

DN: $D \in \# clauses\ S$ **and**

D: $M \models_{as} CNot\ D$ **and**

inv: *all-decomposition-implies-m* N (*get-all-marked-decomposition* M) **and**

state: *state* $S = (M, N, U, k, C)$ **and**

learned-cl: *cdcl_W-learned-clause* S **and**

atm-incl: *no-strange-atm* S

shows *unsatisfiable* (*set-mset* N)

proof (*rule ccontr*)

assume $\neg unsatisfiable\ (set_mset\ N)$

then obtain I **where**

$I: I \models_s set_mset\ N$ **and**

cons: *consistent-interp* I **and**

tot: *total-over-m* I (*set-mset* N)

unfolding *satisfiable-def* **by** *auto*

have *atms-of-msu* $N \cup atms_of_msu\ U = atms_of_msu\ N$

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_W-learned-clause-def* **by** *auto*

ultimately have $I \models D$

using $I\ DN\ cons\ state$ **unfolding** *true-clss-clss-def true-clss-def Ball-def*

by (*metis Un-iff atms-of-msu N atms-of-msu U = atms-of-msu N atms-of-ms-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-ms* (*set-mset* $N \cup (\lambda a. \{\#lit_of\ a\# \}) ' set\ M$) = *atms-of-msu* 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 $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{set-mset } N \models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ‘ set } M$

proof –

have $T: \{\{\# \text{lit-of } L \# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} = \{\}$ **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
 $\text{cdcl}_W: \text{cdcl}_W \ S \ S'$ **and**
 $\text{lev}: \text{cdcl}_W\text{-}M\text{-level-inv } S$ **and**
 $[\text{simp}]: \text{conflicting } S' = C\text{-Clause } \{\#\}$ **and**
 $\text{learned}: \text{cdcl}_W\text{-learned-clause } S$
shows *unsatisfiable* ($\text{set-mset } (\text{init-clss } S)$)
using *assms*

proof –

have $\text{cdcl}_W\text{-learned-clause } S'$ **using** $\text{cdcl}_W\text{-learned-clss } \text{cdcl}_W \ \text{learned } \text{lev}$ **by** *auto*
then have $\text{init-clss } S' \models_{pm} \{\#\}$ **using** *assms*(3) **unfolding** $\text{cdcl}_W\text{-learned-clause-def}$ **by** *auto*
then have $\text{init-clss } S \models_{pm} \{\#\}$
using $\text{cdcl}_W\text{-init-clss}$ [*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 $\text{cdcl}_W \ S \ S'$
and $\text{conflicting } S = C\text{-Clause } \{\#\}$
shows *False*
using *assms* **by** (*induct rule: cdcl_W-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
 $\text{cdcl}_W \ S \ S'$ **and**
 $\text{lev}: \text{cdcl}_W\text{-}M\text{-level-inv } S$ **and**
 $\text{conflicting}: \text{cdcl}_W\text{-conflicting } S$ **and**
 $\text{no-tauto}: \forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$
shows $\forall s \in \# \text{ learned-clss } S'. \neg \text{tautology } s$
using *assms*

proof (*induct rule: cdcl_W-all-induct-lev2*)

case (*backtrack* $K \ i \ M1 \ M2 \ L \ D$) **note** $\text{confl} = \text{this}(3)$
have *consistent-interp* ($\text{lits-of } (\text{trail } S)$) **using** lev **by** (*auto simp: cdcl_W-M-level-inv-decomp*)
moreover
have $\text{trail } S \models_{as} C\text{Not } (D + \{\#L\# \})$
using *conflicting confl* **unfolding** $\text{cdcl}_W\text{-conflicting-def}$ **by** *auto*
then have $\text{lits-of } (\text{trail } S) \models_s C\text{Not } (D + \{\#L\# \})$ **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_W-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_W-state* (*S*:: 'st)
 \longleftrightarrow (*trail* *S* \models_{asm} *init-clss* *S*
 $\vee ((\forall L \in \text{set } (\text{trail } S). \neg \text{is-marked } L) \wedge$
 $(\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} C \text{Not } C)))$

definition *termination-cdcl_W-state* (*S*:: 'st)
 \longleftrightarrow (*trail* *S* \models_{asm} *init-clss* *S*
 $\vee ((\forall L \in \text{atms-of-msu } (\text{init-clss } S). L \in \text{atm-of ' lits-of } (\text{trail } S))$
 $\wedge (\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} C \text{Not } C)))$

17.5 CDCL Strong Completeness

fun *mapi* :: ('a \Rightarrow nat \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a list \Rightarrow 'b list **where**
mapi - - [] = [] |
mapi *f* *n* (*x* # *xs*) = *f* *x* *n* # *mapi* *f* (*n* - 1) *xs*

lemma *mark-not-in-set-mapi[simp]*: $L \notin \text{set } M \implies \text{Marked } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
by (*induct* *M* *arbitrary*: *i*) *auto*

lemma *propagated-not-in-set-mapi[simp]*: $L \notin \text{set } M \implies \text{Propagated } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
by (*induct* *M* *arbitrary*: *i*) *auto*

lemma *image-set-mapi*:
 $f \text{ ' set } (\text{mapi } g \ i \ M) = \text{set } (\text{mapi } (\lambda x \ i. f \ (g \ x \ i)) \ i \ M)$
by (*induction* *M* *arbitrary*: *i*) *auto*

lemma *mapi-map-convert*:
 $\forall x \ i \ j. f \ x \ i = f \ x \ j \implies \text{mapi } f \ i \ M = \text{map } (\lambda x. f \ x \ 0) \ M$
by (*induction* *M* *arbitrary*: *i*) *auto*

lemma *defined-lit-mapi*: $\text{defined-lit } (\text{mapi } \text{Marked } i \ M) \ L \longleftrightarrow \text{atm-of } L \in \text{atm-of ' set } M$
by (*induction* *M*) (*auto simp: defined-lit-map image-set-mapi mapi-map-convert*)

lemma *cdcl_W-can-do-step*:
assumes
consistent-interp (*set* *M*) **and**
distinct *M* **and**
 $\text{atm-of ' } (\text{set } M) \subseteq \text{atms-of-msu } N$
shows $\exists S. \text{rtrancp } \text{cdcl}_W \ (\text{init-state } N) \ S$
 $\wedge \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) \ M, N, \{\#\}, \text{length } M, C\text{-True})$
using *assms*
proof (*induct* *M*)
case *Nil*
then show ?*case* **by** *auto*
next
case (*Cons* *L* *M*) **note** $IH = \text{this}(1)$
have *consistent-interp* (*set* *M*) **and** *distinct* *M* **and** $\text{atm-of ' set } M \subseteq \text{atms-of-msu } N$
using *Cons.prem*s(1-3) **unfolding** *consistent-interp-def* **by** *auto*
then obtain *S* **where**
 $\text{st: } \text{cdcl}_W^{**} \ (\text{init-state } N) \ S \text{ and}$

```

  S: state S = (mapi Marked (length M) M, N, {#}, length M, C-True)
  using IH by auto
let ?S0 = incr-lvl (cons-trail (Marked L (length M + 1)) S)
have undefined-lit (mapi Marked (length M) M) L
  using Cons.premis(1,2) unfolding defined-lit-def consistent-interp-def by fastforce
moreover have init-clss S = N
  using S by blast
moreover have atm-of L ∈ atms-of-msu N using Cons.premis(3) by auto
moreover have undef: undefined-lit (trail S) L
  using S ⟨distinct (L#M)⟩ calculation(1) by (auto simp: defined-lit-mapi defined-lit-map)
ultimately have cdclW S ?S0
  using cdclW.other[OF cdclW-o.decide[OF decide-rule[OF S,
    of L ?S0]]] S by (auto simp: state-eq-def simp del: state-simp)
then show ?case
  using st S undef by (auto intro!: exI[of - ?S0])
qed

```

lemma *cdcl_W-strong-completeness*:

```

  assumes
    set M ⊨s set-mset N and
    consistent-interp (set M) and
    distinct M and
    atm-of ‘ (set M) ⊆ atms-of-msu N
  obtains S where
    state S = (mapi Marked (length M) M, N, {#}, length M, C-True) and
    rtranclp cdclW (init-state N) S and
    final-cdclW-state S
proof –
  obtain S where
    st: rtranclp cdclW (init-state N) S and
    S: state S = (mapi Marked (length M) M, N, {#}, length M, C-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 ⊨asm N using assms(1) true-annots-true-clb 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 *tranclp-conflict-iff[iff]*:

```

  full1 conflict S S' ⟷ conflict S S'
proof –
  have tranclp conflict S S' ⟹ conflict S S'
    unfolding full1-def by (induct rule: tranclp.induct) force+
  then have tranclp conflict S S' ⟹ conflict S S' by (meson rtranclpD)
  then show ?thesis unfolding full1-def by (metis conflictE conflicting-clause.simps(3)
    conflicting-update-conflicting state-eq-conflicting tranclp.intros(1))
qed

```

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

lemma $rtrancpl\text{-}cdcl_W\text{-}cp\text{-}rtrancpl\text{-}cdcl_W$:
 $cdcl_W\text{-}cp^{**}\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
by (*induction rule: rtrancpl-induct*) (*auto simp: cdcl_W-cp.simps dest: cdcl_W.intros*)

lemma $cdcl_W\text{-}cp\text{-}state\text{-}eq\text{-}compatible$:
assumes
 $cdcl_W\text{-}cp\ S\ T$ **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows $cdcl_W\text{-}cp\ S'\ T'$
using *assms*
apply (*induction*)
using *conflict-state-eq-compatible* **apply** *auto[1]*
using *propagate' propagate-state-eq-compatible* **by** *auto*

lemma $trancpl\text{-}cdcl_W\text{-}cp\text{-}state\text{-}eq\text{-}compatible$:
assumes
 $cdcl_W\text{-}cp^{++}\ S\ T$ **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows $cdcl_W\text{-}cp^{++}\ S'\ T'$
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) trancplD*)
then show *?case*
by (*meson cdcl_W-cp-state-eq-compatible rtrancpl.rtrancpl-into-rtrancpl rtrancpl-into-trancpl2 state-eq-ref step(2) step(4) step(5)*)
qed

lemma $conflicting\text{-}clause\text{-}full\text{-}cdcl_W\text{-}cp$:
 $conflicting\ S \neq C\text{-}True \Longrightarrow full\ cdcl_W\text{-}cp\ S\ S$
unfolding *full-def rtrancpl-unfold trancpl-unfold* **by** (*auto simp add: cdcl_W-cp.simps*)

lemma $skip\text{-}unique$:
 $skip\ S\ T \Longrightarrow skip\ S\ T' \Longrightarrow T \sim T'$
by (*fastforce simp: state-eq-def simp del: state-simp*)

lemma $resolve\text{-}unique$:
 $resolve\ S\ T \Longrightarrow resolve\ S\ T' \Longrightarrow T \sim T'$
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* $S = \text{clauses } S'$
using *assms* **by** (*induct rule*: $\text{cdcl}_W\text{-cp.induct}$) (*auto elim!*: $\text{conflictE propagateE}$)

lemma *trancpl-cdcl_W-cp-no-more-clauses*:
assumes $\text{cdcl}_W\text{-cp}^{++} S S'$
shows *clauses* $S = \text{clauses } S'$
using *assms* **by** (*induct rule*: trancpl.induct) (*auto dest*: $\text{cdcl}_W\text{-cp-no-more-clauses}$)⁺

lemma *rtrancpl-cdcl_W-cp-no-more-clauses*:
assumes $\text{cdcl}_W\text{-cp}^{**} S S'$
shows *clauses* $S = \text{clauses } S'$
using *assms* **by** (*induct rule*: rtrancpl.induct) (*fastforce dest*: $\text{cdcl}_W\text{-cp-no-more-clauses}$)⁺

lemma *no-conflict-after-conflict*:
 $\text{conflict } S T \implies \neg \text{conflict } T U$
by *fastforce*

lemma *no-propagate-after-conflict*:
 $\text{conflict } S T \implies \neg \text{propagate } T U$
by *fastforce*

lemma *trancpl-cdcl_W-cp-propagate-with-conflict-or-not*:
assumes $\text{cdcl}_W\text{-cp}^{++} S U$
shows ($\text{propagate}^{++} S U \wedge \text{conflicting } U = C\text{-True}$)
 $\vee (\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U \wedge \text{conflicting } U = C\text{-Clause } D)$
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_W-cp.simps trancpl-into-rtrancpl dest: no-conflict-after-conflict no-propagate-after-conflict)⁺
moreover
have $\text{propagate}^{++} S U \implies \text{conflicting } U = C\text{-True}$
unfolding *trancpl-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_W-cp-conflicting-not-empty[simp]*: $\text{conflicting } S = C\text{-Clause } D \implies \neg \text{cdcl}_W\text{-cp } S S'$
proof
assume $\text{cdcl}_W\text{-cp } S S'$ **and** $\text{conflicting } S = C\text{-Clause } D$
then show *False* **by** (*induct rule*: $\text{cdcl}_W\text{-cp.induct}$) *auto*
qed

lemma *no-step-cdcl_W-cp-no-conflict-no-propagate*:
assumes *no-step cdcl_W-cp* S
shows *no-step conflict* S **and** *no-step propagate* S
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 $\text{cdcl}_W\text{-o } S S'$ and re-apply conflict and propagate *full cdcl_W-cp* $S' S''$

inductive *cdcl_W-stgy* :: $'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**

conflict': $\text{full1 } \text{cdcl}_W\text{-cp } S \ S' \implies \text{cdcl}_W\text{-stgy } S \ S' \mid$
other': $\text{cdcl}_W\text{-o } S \ S' \implies \text{no-step } \text{cdcl}_W\text{-cp } S \implies \text{full } \text{cdcl}_W\text{-cp } S' \ S'' \implies \text{cdcl}_W\text{-stgy } S \ S''$

17.6.2 Invariants

These are the same invariants as before, but lifted

lemma *cdcl_W-cp-learned-clause-inv*:

assumes *cdcl_W-cp* $S \ S'$

shows *learned-clss* $S = \text{learned-clss } S'$

using *assms* **by** (*induct* rule: *cdcl_W-cp.induct*) *fastforce*+

lemma *rtrancpl-cdcl_W-cp-learned-clause-inv*:

assumes *cdcl_W-cp*** $S \ S'$

shows *learned-clss* $S = \text{learned-clss } S'$

using *assms* **by** (*induct* rule: *rtrancpl-induct*) (*fastforce* *dest*: *cdcl_W-cp-learned-clause-inv*)+

lemma *trancpl-cdcl_W-cp-learned-clause-inv*:

assumes *cdcl_W-cp⁺⁺* $S \ S'$

shows *learned-clss* $S = \text{learned-clss } S'$

using *assms* **by** (*simp* *add*: *rtrancpl-cdcl_W-cp-learned-clause-inv* *trancpl-into-rtrancpl*)

lemma *cdcl_W-cp-backtrack-lvl*:

assumes *cdcl_W-cp* $S \ S'$

shows *backtrack-lvl* $S = \text{backtrack-lvl } S'$

using *assms* **by** (*induct* rule: *cdcl_W-cp.induct*) *fastforce*+

lemma *rtrancpl-cdcl_W-cp-backtrack-lvl*:

assumes *cdcl_W-cp*** $S \ S'$

shows *backtrack-lvl* $S = \text{backtrack-lvl } S'$

using *assms* **by** (*induct* rule: *rtrancpl-induct*) (*fastforce* *dest*: *cdcl_W-cp-backtrack-lvl*)+

lemma *cdcl_W-cp-consistent-inv*:

assumes *cdcl_W-cp* $S \ S'$

and *cdcl_W-M-level-inv* S

shows *cdcl_W-M-level-inv* S'

using *assms*

proof (*induct* rule: *cdcl_W-cp.induct*)

case (*conflict'*)

then show ?*case* **using** *cdcl_W-consistent-inv* *cdcl_W.conflict* **by** *blast*

next

case (*propagate'* $S \ S'$)

have *cdcl_W* $S \ S'$

using *propagate'.hyps*(1) *propagate* **by** *blast*

then show *cdcl_W-M-level-inv* S'

using *propagate'.prems*(1) *cdcl_W-consistent-inv* *propagate* **by** *blast*

qed

lemma *full1-cdcl_W-cp-consistent-inv*:

assumes *full1* *cdcl_W-cp* $S \ S'$

and *cdcl_W-M-level-inv* S

shows *cdcl_W-M-level-inv* S'

using *assms* **unfolding** *full1-def*

proof –

have *cdcl_W-cp⁺⁺* $S \ S'$ **and** *cdcl_W-M-level-inv* S **using** *assms* **unfolding** *full1-def* **by** *auto*

then show ?*thesis* **by** (*induct* rule: *trancpl.induct*) (*blast* *intro*: *cdcl_W-cp-consistent-inv*)+

qed

lemma *rtrancpl-cdcl_W-cp-consistent-inv*:
assumes *rtrancpl cdcl_W-cp S S'*
and *cdcl_W-M-level-inv S*
shows *cdcl_W-M-level-inv S'*
using *assms unfolding full1-def*
by (*induction rule: rtrancpl-induct*) (*blast intro: cdcl_W-cp-consistent-inv*)+

lemma *cdcl_W-stgy-consistent-inv*:
assumes *cdcl_W-stgy S S'*
and *cdcl_W-M-level-inv S*
shows *cdcl_W-M-level-inv S'*
using *assms apply (induct rule: cdcl_W-stgy.induct)*
unfolding *full-unfold* **by** (*blast intro: cdcl_W-consistent-inv full1-cdcl_W-cp-consistent-inv cdcl_W.other*)+

lemma *rtrancpl-cdcl_W-stgy-consistent-inv*:
assumes *cdcl_W-stgy** S S'*
and *cdcl_W-M-level-inv S*
shows *cdcl_W-M-level-inv S'*
using *assms by induction (auto dest!: cdcl_W-stgy-consistent-inv)*

lemma *cdcl_W-cp-no-more-init-clss*:
assumes *cdcl_W-cp S S'*
shows *init-clss S = init-clss S'*
using *assms by (induct rule: cdcl_W-cp.induct) auto*

lemma *trancpl-cdcl_W-cp-no-more-init-clss*:
assumes *cdcl_W-cp⁺⁺ S S'*
shows *init-clss S = init-clss S'*
using *assms by (induct rule: trancpl.induct) (auto dest: cdcl_W-cp-no-more-init-clss)*

lemma *cdcl_W-stgy-no-more-init-clss*:
assumes *cdcl_W-stgy S S' and cdcl_W-M-level-inv S*
shows *init-clss S = init-clss S'*
using *assms*
apply (*induct rule: cdcl_W-stgy.induct*)
unfolding *full1-def full-def* **apply** (*blast dest: trancpl-cdcl_W-cp-no-more-init-clss trancpl-cdcl_W-o-no-more-init-clss*)
by (*metis cdcl_W-o-no-more-init-clss rtrancpl-unfold trancpl-cdcl_W-cp-no-more-init-clss*)

lemma *rtrancpl-cdcl_W-stgy-no-more-init-clss*:
assumes *cdcl_W-stgy** S S' and cdcl_W-M-level-inv S*
shows *init-clss S = init-clss S'*
using *assms*
apply (*induct rule: rtrancpl-induct, simp*)
using *cdcl_W-stgy-no-more-init-clss* **by** (*simp add: rtrancpl-cdcl_W-stgy-consistent-inv*)

lemma *cdcl_W-cp-dropWhile-trail'*:
assumes *cdcl_W-cp S S'*
obtains *M where trail S' = M @ trail S and (∀ l ∈ set M. ¬is-marked l)*
using *assms by induction fastforce+*

lemma *rtrancpl-cdcl_W-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!*: $cdcl_W\text{-cp-dropWhile-trail}$) $+$

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 $rtrancp\text{-}cdcl_W\text{-cp-dropWhile-trail}$:

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*: $cdcl_W\text{-cp-dropWhile-trail}$) $+$

This theorem can be seen as 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**
 $finite (atms\text{-of-msu } (init\text{-class } S))$
shows $length (trail S) \leq card (atms\text{-of-msu } (init\text{-class } S))$

proof –

obtain $M N U k D$ **where** S : $state S = (M, N, U, k, D)$ **by** (*cases state S, auto*)
have $finite (atm\text{-of } 'lits\text{-of } (trail S))$
using *assms*(1,3) **unfolding** S **by** (*auto simp add: finite-subset*)
have $length (trail S) = card (atm\text{-of } 'lits\text{-of } (trail S))$
using $no\text{-dup-length-eq-card-atm-of-lits-of no-d}$ **by** *blast*
then show $?thesis$ **using** *assms*(1) **unfolding** $no\text{-strange-atm-def}$
by (*auto simp add: assms*(3) *card-mono*)

qed

lemma $cdcl_W\text{-cp-decreasing-measure}$:

assumes
 $cdcl_W$: $cdcl_W\text{-cp } S T$ **and**
 $M\text{-lev}$: $cdcl_W\text{-M-level-inv } S$ **and**
 $alien$: $no\text{-strange-atm } S$
shows $(\lambda S. card (atms\text{-of-msu } (init\text{-class } S)) - length (trail S))$
 $+ (if \text{ conflicting } S = C\text{-True then } 1 \text{ else } 0)) S$
 $> (\lambda S. card (atms\text{-of-msu } (init\text{-class } S)) - length (trail S))$
 $+ (if \text{ conflicting } S = C\text{-True then } 1 \text{ else } 0)) T$

using *assms*

proof –

have $length (trail T) \leq card (atms\text{-of-msu } (init\text{-class } T))$
apply (*rule length-model-le-vars*)
using $cdcl_W\text{-no-strange-atm-inv alien M-lev}$ **apply** (*meson cdcl_W cdcl_W.simps cdcl_W-cp.cases*)
using $M\text{-lev cdcl_W cdcl_W-cp-consistent-inv cdcl_W-M-level-inv-def}$ **apply** *blast*
using $cdcl_W$ **by** (*auto simp: cdcl_W-cp.simps*)
with *assms*
show $?thesis$ **by induction** (*auto split: split-if-asm*) $+$

qed

lemma $cdcl_W\text{-cp-wf}$: $wf \{(b,a). (cdcl_W\text{-M-level-inv } a \wedge no\text{-strange-atm } a)$

$\wedge cdcl_W\text{-cp } a b\}$

apply (*rule wf-wf-if-measure'[of less-than - -*

```

    (λS. card (atms-of-msu (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 rtrancpl-cdclW-all-struct-inv-cdclW-cp-iff-rtrancpl-cdclW-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 cdclW** S T
      by (metis rtrancpl-unfold cdclW-cp-conflicting-not-empty cp st
        rtrancpl-propagate-is-rtrancpl-cdclW trancpl-cdclW-cp-propagate-with-conflict-or-not)
    then have
      cdclW-M-level-inv T and
      no-strange-atm T
      using ⟨cdclW** S T⟩ apply (simp add: asms(1) rtrancpl-cdclW-consistent-inv)
      using ⟨cdclW** S T⟩ alien rtrancpl-cdclW-no-strange-atm-inv lev by blast
    then have (λa b. (cdclW-M-level-inv a ∧ no-strange-atm a)
      ∧ cdclW-cp a b)** T U
      using cp by auto
    then show ?case using IH by auto
  qed
qed

lemma cdclW-cp-normalized-element:
  assumes
    lev: cdclW-M-level-inv S and
    no-strange-atm S
  obtains T where full cdclW-cp S T
proof -
  let ?inv = λa. (cdclW-M-level-inv a ∧ no-strange-atm a)
  obtain T where T: full (λa b. ?inv a ∧ cdclW-cp a b) S T
  using cdclW-cp-wf wf-exists-normal-form[of λa b. ?inv a ∧ cdclW-cp a b]
  unfolding full-def by blast
  then have cdclW-cp** S T
    using rtrancpl-cdclW-all-struct-inv-cdclW-cp-iff-rtrancpl-cdclW-cp asms unfolding full-def
    by blast
  moreover
    then have cdclW** 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 $\langle \text{cdcl}_W^{**} S T \rangle$ apply (*simp add: assms(1) rtranclp-cdcl_W-consistent-inv*)
 using $\langle \text{cdcl}_W^{**} S T \rangle$ 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-ms:*
 $C + \{\#L\# \} \in \# A \implies x \in \text{atms-of } C \implies x \in \text{atms-of-msu } A$
 by (*metis add.commute atm-iff-pos-or-neg-lit atms-of-atms-of-ms-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-ms*
 in-atms-of-implies-atm-of-on-atms-of-ms)

lemma *always-exists-full-cdcl_W-cp-step:*
 assumes *no-strange-atm S*
 shows $\exists S''. \text{full } \text{cdcl}_W\text{-cp } S S''$
 using *assms*
proof (*induct card (atms-of-msu (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-msu (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_W-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 $C + \{\#L\# \} \in \# \text{clauses } S$
 and $M \models_{\text{as}} C \text{Not } C$
 and *undefined-lit M L*
 using *propagate* by *auto*
 have *atms-of-msu U \subseteq atms-of-msu N* using *alien S unfolding no-strange-atm-def* by *auto*
 then have *atm-of L \in atms-of-msu (init-clss S)*
 using $\langle C + \{\#L\# \} \in \# \text{clauses } S \rangle$ *S unfolding atms-of-ms-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_W-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':  $\text{conflict } S S'$  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'  $\text{cdcl}_W\text{-cp.conflict'}$  by blast
}
moreover {
  assume a:  $\exists S'. \text{propagate } S S'$ 
  then obtain S' where propagate:  $\text{propagate } S S'$  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-stange-atm by blast
    then have atm-of L  $\in$  atms-of-msu N using S' unfolding no-strange-atm-def by auto
    then have  $\bigwedge A. \{\text{atm-of } L\} \subseteq \text{atms-of-msu } N - A \vee \text{atm-of } L \in A$  by force
  moreover have Suc n - card {atm-of L} = n by simp
  moreover have card (atms-of-msu N - atm-of ' lits-of M) = Suc n
    using card S S' by simp
  ultimately
    have card (atms-of-msu 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-msu (init-clss S') - atm-of ' lits-of (trail S'))
      using card S S' by simp
  then have a1: Ex (full  $\text{cdcl}_W\text{-cp } S'$ ) using IH <no-strange-atm S'> by blast
  have ?case
    proof -
      obtain S'' :: 'st where
        ff1:  $\text{cdcl}_W\text{-cp}^{**} S' S'' \wedge \text{no-step } \text{cdcl}_W\text{-cp } S''$ 
        using a1 unfolding full-def by blast
      have  $\text{cdcl}_W\text{-cp}^{**} S S''$ 
        using ff1  $\text{cdcl}_W\text{-cp.intros}(2)[OF \text{propagate}]$ 
        by (metis (no-types) converse-rtranclp-into-rtranclp)
      then have  $\exists S''. \text{cdcl}_W\text{-cp}^{**} S S'' \wedge (\forall S'''. \neg \text{cdcl}_W\text{-cp } S'' S''')$ 
        using ff1 by blast
      then show ?thesis unfolding full-def
        by meson
    qed
  }
  ultimately show ?case unfolding full-def by (metis  $\text{cdcl}_W\text{-cp.cases } rtranclp.rtrancl\text{-refl}$ )
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 $\text{no-clause-is-false} :: 'st \Rightarrow \text{bool}$ **where**
 $\text{no-clause-is-false} \equiv$

$\lambda S. (\text{conflicting } S = C\text{-True} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{as} C\text{Not } D))$

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_W-cp-not-any-negated-init-clss*:

assumes *full cdcl_W-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_W-cp-not-any-negated-init-clss*:

assumes *full1 cdcl_W-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_W-stgy-not-non-negated-init-clss*:

assumes *cdcl_W-stgy* $S S'$
shows *no-clause-is-false* S'
using *assms apply* (*induct rule: cdcl_W-stgy.induct*)
using *full1-cdcl_W-cp-not-any-negated-init-clss* *full-cdcl_W-cp-not-any-negated-init-clss* **by** *metis+*

lemma *rtranclp-cdcl_W-stgy-not-non-negated-init-clss*:

assumes *cdcl_W-stgy*** $S S'$ **and** *no-clause-is-false* S
shows *no-clause-is-false* S'
using *assms* **by** (*induct rule: rtranclp-induct*) (*auto simp: cdcl_W-stgy-not-non-negated-init-clss*)

lemma *cdcl_W-stgy-conflict-ex-lit-of-max-level*:

assumes *cdcl_W-cp* $S S'$
and *no-clause-is-false* S
and *cdcl_W-M-level-inv* S
shows *conflict-is-false-with-level* S'
using *assms*

proof (*induct rule: cdcl_W-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 *rtranclp-cdcl_W-cp-propa-or-propa-confl*:

assumes *cdcl_W-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 rtrancp-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: cdclW-M-level-inv S
  shows conflict-is-false-with-level U
proof (intro allI impI)
  fix D
  assume confl: conflicting U = C-Clause D and
    D: D  $\neq$  {#}
  consider (CT) conflicting S = C-True | (SD) D' where conflicting S = C-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) cdclW-cp-conflicting-not-empty full-def rtrancpD trancpD)
    then show ?thesis using assms(3) confl D by blast-
  next
  case CT
  have init-clss U = init-clss S and learned-clss U = learned-clss S
  using assms(1) unfolding full-def
  apply (metis (no-types) rtrancpD trancp-cdclW-cp-no-more-init-clss)
  by (metis (mono-tags, lifting) assms(1) full-def rtrancp-cdclW-cp-learned-clause-inv)
  obtain T where propagate** S T and TU: conflict T U
  proof -
    have f5: U  $\neq$  S
    using confl CT by force
    then have cdclW-cp++ S U
    by (metis full full-def rtrancpD)
    have  $\bigwedge p \text{ pa. } \neg \text{propagate } p \text{ pa} \vee \text{conflicting } \text{pa} =$ 
      (C-True::'v literal multiset conflicting-clause)
    by auto
    then show ?thesis
    using f5 that trancp-cdclW-cp-propagate-with-conflict-or-not[OF  $\langle \text{cdcl}_W\text{-cp}^{++} S U \rangle$ ]
  qed

```

```

    full confl CT unfolding full-def by auto
qed
have init-clss T = init-clss S and learned-clss T = learned-clss S
  using TU ⟨init-clss U = init-clss S⟩ ⟨learned-clss U = learned-clss S⟩ by auto
then have D ∈# clauses S
  using TU confl by (fastforce simp: clauses-def)
then have ¬ trail S ⊨as CNot D
  using cls-f CT by simp
moreover
  obtain M where tr-U: trail U = M @ trail S and nm: ∀ m ∈ set M. ¬ is-marked m
    by (metis (mono-tags, lifting) assms(1) full-def rtrancpl-cdclW-cp-dropWhile-trail)
  have trail U ⊨as CNot D
    using TU confl by auto
ultimately obtain L where L ∈# D and ¬L ∈ lits-of M
  unfolding tr-U CNot-def true-annot-def Ball-def true-annot-def true-cls-def by auto

moreover have inv-U: cdclW-M-level-inv U
  by (metis cdclW-stgy.conflict' cdclW-stgy-consistent-inv full full-unfold lev)
moreover
  have backtrack-lvl U = backtrack-lvl S
    using full unfolding full-def by (auto dest: rtrancpl-cdclW-cp-backtrack-lvl)

moreover
  have no-dup (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' ≡

∀ D M1 M2 L. M1 @ Propagated L D # M2 = trail S' ⟶ D - {#L#} ≠ {#}
 ⟶ (∃ L. L ∈ # D ∧ get-level L (trail S') = get-maximum-possible-level M1)

definition *no-more-propagation-to-do* :: 'st ⇒ bool **where**

no-more-propagation-to-do S ≡

∀ D M M' L. D + {#L#} ∈ # clauses S ⟶ trail S = M' @ M ⟶ M ⊨_{as} CNot D
 ⟶ undefined-lit M L ⟶ get-maximum-possible-level M < backtrack-lvl S
 ⟶ (∃ L. L ∈ # D ∧ get-level L (trail S) = 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_W-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#} ∈ # clauses S **and**

M ⊨_{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'#} ∈ # clauses S'

and trail S' = M2 @ M1

and get-max: get-maximum-possible-level M1 < backtrack-lvl S'

and M1 ⊨_{as} CNot D

and undef: undefined-lit M1 L'

have tl M2 @ M1 = trail S ∨ (M2 = [] ∧ M1 = Propagated L ((C + {#L#})) # M)

using (trail S' = M2 @ M1) S' S **by** (cases M2) *auto*

moreover {

```

assume  $tl\ M2 @\ M1 = trail\ S$ 
moreover have  $D + \{\#L'\#\} \in \# \text{ clauses } S$  using  $D-L\ S\ S'$  unfolding  $\text{clauses-def}$  by  $auto$ 
moreover have  $get\text{-}maximum\text{-}possible\text{-}level\ M1 < backtrack\text{-}lvl\ S$ 
  using  $get\text{-}max\ S\ S'$  by  $auto$ 
ultimately obtain  $L'$  where  $L' \in \# D$  and
   $get\text{-}level\ L' (trail\ S) = get\text{-}maximum\text{-}possible\text{-}level\ M1$ 
  using  $H\ \langle M1 \models_{as} CNot\ D \rangle$  undef unfolding  $no\text{-}more\text{-}propagation\text{-}to\text{-}do\text{-}def$  by  $metis$ 
moreover
  { have  $cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$ 
    using  $cdcl_W\text{-}consistent\text{-}inv[OF - M]\ cdcl_W.propagate[OF\ propagate]$  by  $blast$ 
    then have  $no\text{-}dup\ ?M'$  using  $S'$  unfolding  $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$  by  $auto$ 
    moreover
      have  $atm\text{-}of\ L' \in atm\text{-}of\ ' (lits\text{-}of\ M1)$ 
      using  $\langle L' \in \# D \rangle\ \langle M1 \models_{as} CNot\ D \rangle$  by  $(metis\ atm\text{-}of\text{-}uminus\ image\text{-}eqI\ in\ CNot\text{-}implies\text{-}uminus(2))$ 
      then have  $atm\text{-}of\ L' \in atm\text{-}of\ ' (lits\text{-}of\ M)$ 
      using  $\langle tl\ M2 @\ M1 = trail\ S \rangle\ S$  by  $auto$ 
      ultimately have  $atm\text{-}of\ L \neq atm\text{-}of\ L'$  unfolding  $lits\text{-}of\text{-}def$  by  $auto$ 
    }
  ultimately have  $\exists L' \in \# D. get\text{-}level\ L' (trail\ S') = get\text{-}maximum\text{-}possible\text{-}level\ M1$ 
  using  $S\ S'$  by  $auto$ 
}
moreover {
  assume  $M2 = []$  and  $M1: M1 = Propagated\ L\ ( (C + \{\#L'\#\}) \# M$ 
  have  $cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$ 
  using  $cdcl_W\text{-}consistent\text{-}inv[OF - M]\ cdcl_W.propagate[OF\ propagate]$  by  $blast$ 
  then have  $get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (trail\ S') = rev\ ([Suc\ 0..<(Suc\ 0+k)])$ 
  using  $S'$  unfolding  $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$  by  $auto$ 
  then have  $get\text{-}maximum\text{-}possible\text{-}level\ M1 = backtrack\text{-}lvl\ S'$ 
  using  $get\text{-}maximum\text{-}possible\text{-}level\text{-}max\text{-}get\text{-}all\text{-}levels\text{-}of\text{-}marked[of\ M1]\ S'\ M1$ 
  by  $(auto\ intro: Max\text{-}eqI)$ 
  then have  $False$  using  $get\text{-}max$  by  $auto$ 
}
ultimately show  $\exists L. L \in \# D \wedge get\text{-}level\ L (trail\ S') = get\text{-}maximum\text{-}possible\text{-}level\ M1$  by  $fast$ 
qed
qed

```

lemma *conflict-no-more-propagation-to-do:*

```

assumes  $conflict: conflict\ S\ S'$ 
and  $H: no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S$ 
and  $M: cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ 
shows  $no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S'$ 
using  $assms$  unfolding  $no\text{-}more\text{-}propagation\text{-}to\text{-}do\text{-}def\ conflict.simps$  by  $force$ 

```

lemma *cdcl_W-cp-no-more-propagation-to-do:*

```

assumes  $conflict: cdcl_W\text{-}cp\ S\ S'$ 
and  $H: no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S$ 
and  $M: cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ 
shows  $no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S'$ 
using  $assms$ 
proof  $(induct\ rule: cdcl_W\text{-}cp.induct)$ 
case  $(conflict'\ S\ S')$ 
then show  $?case$  using  $conflict\text{-}no\text{-}more\text{-}propagation\text{-}to\text{-}do[of\ S\ S']$  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 *cdcl_W-then-exists-cdcl_W-stgy-step*:

assumes

o: *cdcl_W-o* S S' **and**
alien: *no-strange-atm* S **and**
lev: *cdcl_W-M-level-inv* S

shows $\exists S'. \text{cdcl}_W\text{-stgy } S S'$

proof –

obtain S'' **where** *full cdcl_W-cp* $S' S''$

using *always-exists-full-cdcl_W-cp-step* *alien cdcl_W-no-strange-atm-inv cdcl_W-o-no-more-init-clss*
o other lev **by** (*meson cdcl_W-consistent-inv*)

then show *?thesis*

using *assms* **by** (*metis always-exists-full-cdcl_W-cp-step cdcl_W-stgy.conflict' full-unfold other'*)

qed

lemma *backtrack-no-decomp*:

assumes S : *state* $S = (M, N, U, k, C\text{-Clause } (D + \{\#L\#}))$

and L : *get-level* $L M = k$

and D : *get-maximum-level* $D M < k$

and $M-L$: *cdcl_W-M-level-inv* S

shows $\exists S'. \text{cdcl}_W\text{-o } S S'$

proof –

have $L-D$: *get-level* $L M = \text{get-maximum-level } (D + \{\#L\#}) M$

using $L D$ **by** (*simp add: get-maximum-level-plus*)

let $?i = \text{get-maximum-level } D M$

obtain $K M1 M2$ **where** K : (*Marked* K ($?i + 1$) $\# M1, M2$) $\in \text{set } (\text{get-all-marked-decomposition } M)$

using *backtrack-ex-decomp*[*OF* $M-L$, *of* $?i$] $D S$ **by** *auto*

show *?thesis* **using** *backtrack-rule*[*OF* $S K L L-D$] **by** (*meson bj cdcl_W-bj.simps state-eq-ref*)

qed

lemma *cdcl_W-stgy-final-state-conclusive*:

assumes *termi*: $\forall S'. \neg \text{cdcl}_W\text{-stgy } S S'$

and *decomp*: *all-decomposition-implies-m* (*init-clss* S) (*get-all-marked-decomposition* (*trail* S))

and *learned*: *cdcl_W-learned-clause* S

and *level-inv*: *cdcl_W-M-level-inv* S

and *alien*: *no-strange-atm* S

and *no-dup*: *distinct-cdcl_W-state* S

and *confl*: *cdcl_W-conflicting* S

and *confl-k*: *conflict-is-false-with-level* S

shows (*conflicting* $S = C\text{-Clause } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$)

$\vee (\text{conflicting } S = C\text{-True} \wedge \text{trail } S \models_{\text{as set-mset}} (\text{init-clss } S))$

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 *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 (*case-tac conflicting* S , *auto*)

by (*case-tac x2*, *auto*)


```

moreover {
  assume conflicting S = C-Clause {#}
  then have unsatisfiable (set-mset (init-clss S))
    using assms(3) unfolding cdclW-learned-clause-def true-clss-cls-def
    by (metis (no-types, lifting) Un-insert-right atms-of-empty satisfiable-def
        sup-bot.right-neutral total-over-m-insert total-over-set-empty true-clss-empty)
}
moreover {
  assume conflicting S = C-True
  { assume ¬?M ⊨asm ?N
    have atm-of ' (lits-of ?M) = atms-of-msu ?N (is ?A = ?B)
    proof
      show ?A ⊆ ?B using alien unfolding no-strange-atm-def by auto
      show ?B ⊆ ?A
        proof (rule ccontr)
          assume ¬?B ⊆ ?A
          then obtain l where l ∈ ?B and l ∉ ?A by auto
          then have undefined-lit ?M (Pos l)
            using ⟨l ∉ ?A⟩ unfolding lits-of-def by (auto simp add: defined-lit-map)
          then have ∃ S'. cdclW-o S S'
            using cdclW-o.decide decide.intros ⟨l ∈ ?B⟩ no-strange-atm-def
            by (metis ⟨conflicting S = C-True⟩ literal.sel(1) state-eq-def)
          then show False
            using termi cdclW-then-exists-cdclW-stgy-step[OF - alien] level-inv by blast
        qed
      qed
    obtain D where ¬ ?M ⊨a D and D ∈# ?N
      using ⟨¬?M ⊨asm ?N⟩ unfolding lits-of-def true-annots-def Ball-def by auto
    have atms-of D ⊆ atm-of ' (lits-of ?M)
      using ⟨D ∈# ?N⟩ unfolding ⟨atm-of ' (lits-of ?M) = atms-of-msu ?N⟩ atms-of-ms-def
      by (auto simp add: atms-of-def)
    then have a1: atm-of ' set-mset D ⊆ atm-of ' lits-of (trail S)
      by (auto simp add: atms-of-def lits-of-def)
    have total-over-m (lits-of ?M) {D}
      using ⟨atms-of D ⊆ atm-of ' (lits-of ?M)⟩ atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      by (fastforce simp: total-over-set-def)
    then have ?M ⊨as CNot D
      using total-not-true-clss-true-clss-CNot ⟨¬ trail S ⊨a D⟩ true-annot-def
      true-annots-true-clss by fastforce
    then have False
      proof -
        obtain S' where
          f2: full cdclW-cp S S'
          by (meson alien always-exists-full-cdclW-cp-step level-inv)
        then have S' = S
          using cdclW-stgy.conflict'[of S] by (metis (no-types) full-unfold termi)
        then show ?thesis
          using f2 ⟨D ∈# init-clss S⟩ ⟨conflicting S = C-True⟩ ⟨trail S ⊨as CNot D⟩
          clauses-def full-cdclW-cp-not-any-negated-init-clss by auto
      qed
    qed
  }
  then have ?M ⊨asm ?N by blast
}
moreover {
  assume ∃ D L. conflicting S = C-Clause (D + {#L#})

```

obtain $D \ L$ **where** LD : *conflicting* $S = C\text{-Clause } (D + \{\#L\# \})$ **and** $\text{get-level } L \ ?M = ?k$
proof –
obtain $mm :: 'v \text{ literal multiset}$ **and** $ll :: 'v \text{ literal}$ **where**
 $f2$: *conflicting* $S = C\text{-Clause } (mm + \{\#ll\# \})$
using $\langle \exists D \ L. \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \}) \rangle$ **by force**
have $\forall m. (\text{conflicting } S \neq C\text{-Clause } m \vee m = \{\#\})$
 $\vee (\exists l. l \in \# \ m \wedge \text{get-level } l \ (\text{trail } S) = \text{backtrack-lvl } S)$
using *confl-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-conflicting-def* **by auto**
then have $?M \neq []$ **unfolding** *true-annots-def* *Ball-def* *true-annot-def* *true-cls-def* **by force**
{ have M : $?M = \text{hd } ?M \ \# \ \text{tl } ?M$ **using** $\langle ?M \neq [] \rangle$ *list.collapse* **by fastforce**
assume *marked*: *is-marked* ($\text{hd } ?M$)
then obtain k' **where** $k': k' + 1 = ?k$
using *level-inv* M **unfolding** *cdcl_W-M-level-inv-def*
by (*cases* $\text{hd } (\text{trail } S)$; *cases* $\text{trail } S$) **auto**
obtain $L' \ l'$ **where** L' : $\text{hd } ?M = \text{Marked } L' \ l'$ **using** *marked* **by** (*case-tac* $\text{hd } ?M$) **auto**
have *get-all-levels-of-marked* ($\text{hd } (\text{trail } S) \ \# \ \text{tl } (\text{trail } S)$)
 $= \text{rev } [1..<1 + \text{length } (\text{get-all-levels-of-marked } ?M)]$
using *level-inv* $\langle \text{get-level } L \ ?M = ?k \rangle \ M$ **unfolding** *cdcl_W-M-level-inv-def* $M[\text{symmetric}]$
by blast
then have $l' \text{-tl}$: $l' \ \# \ \text{get-all-levels-of-marked } (\text{tl } ?M)$
 $= \text{rev } [1..<1 + \text{length } (\text{get-all-levels-of-marked } ?M)]$ **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 *Suc-le-mono* **calculation by** (*fastforce* *simp* *add*: *upt.simps(2)*)
finally have
 $l' = ?k$ **and**
 $g\text{-r}$: *get-all-levels-of-marked* ($\text{tl } (\text{trail } S)$)
 $= \text{rev } [1..<\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$
using *level-inv* $\langle \text{get-level } L \ ?M = ?k \rangle \ M$ **unfolding** *cdcl_W-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 (*metis* *atm-of-uminus* *imageI*)
have $L' = -L$
proof (*rule ccontr*)
assume $\neg ?thesis$
moreover have $-L \in \text{lits-of } ?M$ **using** *confl* LD **unfolding** *cdcl_W-conflicting-def* **by auto**
ultimately have $\text{get-level } L \ (\text{hd } (\text{trail } S) \ \# \ \text{tl } (\text{trail } S)) = \text{get-level } L \ (\text{tl } ?M)$
using *cdcl_W-M-level-inv-decomp(1)* [*OF level-inv*] **unfolding** L' *consistent-interp-def*
by (*metis* (*no-types*, *lifting*) $L' \ M$ *atm-of-eq-atm-of* *get-level-skip-beginning* *insert-iff* *lits-of-cons* *marked-lit.sel(1)*)
moreover
have $\text{length } (\text{get-all-levels-of-marked } (\text{trail } S)) = ?k$
using *level-inv* **unfolding** *cdcl_W-M-level-inv-def* **by auto**
then have $\text{Max } (\text{set } (0 \ \# \ \text{get-all-levels-of-marked } (\text{tl } (\text{trail } S)))) = ?k - 1$
unfolding $g\text{-r}$ **by** (*auto* *simp* *add*: *Max-n-upt*)
then have $\text{get-level } L \ (\text{tl } ?M) < ?k$
using *get-maximum-possible-level-ge-get-level* [*of* $L \ \text{tl } ?M$]
by (*metis* *One-nat-def* *add.right-neutral* *add-Suc-right* *diff-add-inverse2*)

```

    get-maximum-possible-level-max-get-all-levels-of-marked k' le-imp-less-Suc
    list.simps(15))
  finally show False using ⟨get-level L ?M = ?k⟩ M by auto
qed
have L: hd ?M = Marked (-L) ?k using ⟨l' = ?k⟩ ⟨L' = -L⟩ L' by auto

have g-a-l: get-all-levels-of-marked ?M = rev [1..<1 + ?k]
  using level-inv ⟨get-level L ?M = ?k⟩ M unfolding cdclW-M-level-inv-def by auto
have g-k: get-maximum-level D (trail S) ≤ ?k
  using get-maximum-possible-level-ge-get-maximum-level[of D ?M]
  get-maximum-possible-level-max-get-all-levels-of-marked[of ?M]
  by (auto simp add: Max-n-upt g-a-l)
have get-maximum-level D (trail S) < ?k
  proof (rule ccontr)
    assume ¬ ?thesis
    then have get-maximum-level D (trail S) = ?k using M g-k unfolding L by auto
    then obtain L' where L' ∈# D and L-k: get-level L' ?M = ?k
      using get-maximum-level-exists-lit[of ?k D ?M] unfolding k'[symmetric] by auto
    have L ≠ L' using no-dup ⟨L' ∈# D⟩
      unfolding distinct-cdclW-state-def LD by (metis add.commute add-eq-self-zero
        count-single count-union less-not-refl3 distinct-mset-def union-single-eq-member)
    have L' = -L
      proof (rule ccontr)
        assume ¬ ?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  $-L' \notin \# ?D$ 
  then have False
    using bj[OF cdclW-bj.skip[OF skip-rule[OF  $-L' \notin \# ?D$   $\langle ?D \neq \{\#\} \rangle$ , 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  $-L' \in \# ?D$ 
  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.. $\leq$  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  $L' \ C \ \# \ tl \ ?M$ )  $\leq$  ?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
     $\vee$  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-D'-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 cdclW-o S (update-conflicting (C-Clause ( $D' \ \# \cup \ C'$ )) (tl-trail S))
          using f1 bj[OF cdclW-bj.resolve[OF resolve-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 cdclW-then-exists-cdclW-stgy-step termi level-inv)
      qed
    }
  moreover {
    assume get-maximum-level  $D'$  (Propagated  $L' \ C \ \# \ tl \ ?M$ ) < ?k
    then have False
      proof -
        assume a1: get-maximum-level  $D'$  (Propagated  $L' \ C \ \# \ tl \ (trail \ S)$ ) < backtrack-lvl S
        obtain mm :: 'v literal multiset and ll :: 'v literal where
          f2: conflicting S = C-Clause (mm +  $\{\#ll\#\}$ )
          get-level ll (trail S) = backtrack-lvl S
          using LD (get-level L (trail S) = backtrack-lvl S) by blast
        then have f3: get-maximum-level  $D'$  (trail S)  $\leq$  get-level ll (trail S)
          using  $M$  a1 by force
        have get-level ll (trail S)  $\neq$  get-maximum-level  $D'$  (trail S)
          using f2  $M$  calculation(2) by presburger
        have f1: trail S = Propagated  $L' \ C \ \# \ tl \ (trail \ S)$ 
          conflicting S = C-Clause ( $D' + \{\#-L'\#\}$ )
          using  $D' \ LD \ M$  by force+
        have f2: conflicting S = C-Clause (mm +  $\{\#ll\#\}$ )
          get-level ll (trail S) = backtrack-lvl S

```

```

      using f2 by force+
    have ll = - L'
    by (metis (no-types) D' LD (get-level ll (trail S) ≠ get-maximum-level D' (trail S))
        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 cdclW-then-exists-cdclW-stgy-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 cdclW-cp-tranclp-cdclW:
  cdclW-cp S S' ⇒ cdclW++ S S'
  apply (induct rule: cdclW-cp.induct)
  by (meson cdclW.conflict cdclW.propagate tranclp.r-into-trancl tranclp.trancl-into-trancl)+

```

```

lemma tranclp-cdclW-cp-tranclp-cdclW:
  cdclW-cp++ S S' ⇒ cdclW++ S S'
  apply (induct rule: tranclp.induct)
  apply (simp add: cdclW-cp-tranclp-cdclW)
  by (meson cdclW-cp-tranclp-cdclW tranclp-trans)

```

```

lemma cdclW-stgy-tranclp-cdclW:
  cdclW-stgy S S' ⇒ cdclW++ S S'
proof (induct rule: cdclW-stgy.induct)
  case conflict'
  then show ?case
    unfolding full1-def by (simp add: tranclp-cdclW-cp-tranclp-cdclW)
next
  case (other' S' S'')
  then have S' = S'' ∨ cdclW-cp++ S' S''
    by (simp add: rtranclp-unfold full-def)
  then show ?case
    using other' by (meson cdclW-ops.other cdclW-ops-axioms tranclp.r-into-trancl
        tranclp-cdclW-cp-tranclp-cdclW tranclp-trans)
qed

```

```

lemma tranclp-cdclW-stgy-tranclp-cdclW:
  cdclW-stgy++ S S' ⇒ cdclW++ S S'
  apply (induct rule: tranclp.induct)
  using cdclW-stgy-tranclp-cdclW apply blast
  by (meson cdclW-stgy-tranclp-cdclW tranclp-trans)

```

```

lemma rtranclp-cdclW-stgy-rtranclp-cdclW:
  cdclW-stgy** S S' ⇒ cdclW** S S'
  using rtranclp-unfold[of cdclW-stgy S S'] tranclp-cdclW-stgy-tranclp-cdclW[of S S'] by auto

```

lemma *cdcl_W-o-conflict-is-false-with-level-inv*:

assumes

cdcl_W-o S S' and

lev: cdcl_W-M-level-inv S and

confl-inv: conflict-is-false-with-level S and

n-d: distinct-cdcl_W-state S and

conflicting: cdcl_W-conflicting S

shows *conflict-is-false-with-level S'*

using *assms(1,2)*

proof (*induct rule: cdcl_W-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-cdcl_W-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 cdcl_W-conflicting-def* **by** *auto*

ultimately **have** $-L \in \text{lits-of } (\text{Propagated L } (C + \{\#L\#\})) \# M$

using *in-CNot-implies-uminus(2)* **by** *blast*

moreover **have** *no-dup (Propagated L (C + {#L#})) # M*

using *lev tr-S unfolding cdcl_W-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-D: get-maximum-level D (Propagated L (C + {#L#})) # M*

$= \text{get-maximum-level } D \ M$

proof $-$

have $\forall a f L. ((a::'v) \in f \ 'L) = (\exists l. (l::'v \text{ literal}) \in L \wedge a = f l)$

by *blast*

then **show** *?thesis*

using *get-maximum-level-skip-first[of L D (C + {#L#}) M] unfolding atms-of-def*

by (*metis (no-types) $\hookleftarrow - L \notin\# D \hookrightarrow L \notin\# D \hookrightarrow \text{atm-of-eq-atm-of mem-set-mset-iff}$*)

qed

{ assume

get-maximum-level D (Propagated L (C + {#L#})) # M = backtrack-lvl S and

backtrack-lvl S > 0

then **have** *D: get-maximum-level D M = backtrack-lvl S* **unfolding** *g-D* **by** *blast*

then **have** *?case*

using *tr-S $\hookleftarrow \text{backtrack-lvl } S > 0 \hookrightarrow \text{get-maximum-level-exists-lit[of backtrack-lvl S D M]$ T*

by *auto*

}

moreover **{**

assume [*simp*]: *backtrack-lvl S = 0*

have $\bigwedge L. \text{get-level } L \ M = 0$

proof $-$

fix *L*

have *atm-of L \notin atm-of ' (lits-of M) $\implies \text{get-level } L \ M = 0$* **by** *auto*

moreover **{**

assume *atm-of L \in atm-of ' (lits-of M)*

have *g-r: get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{backtrack-lvl } S)$]*

using *lev tr-S unfolding cdcl_W-M-level-inv-def* **by** *auto*

have *Max (insert 0 (set (get-all-levels-of-marked M))) = (backtrack-lvl S)*

unfolding *g-r* **by** (*simp add: Max-n-upt*)

}

```

    then have get-level  $L\ M = 0$ 
      using get-maximum-possible-level-ge-get-level[of  $L\ M$ ]
      unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
    }
    ultimately show get-level  $L\ M = 0$  by blast
  qed
  then have ?case using get-maximum-level-exists-lit-of-max-level[of  $D\ \# \cup C\ M$ ] tr-S\ T
    by (auto simp: Bex-mset-def)
}
ultimately show ?case using resolve.hyps(3) by blast
next
case (skip  $L\ C'\ M\ D\ T$ ) note tr-S = this(1) and  $D = \text{this}(2)$  and  $T = \text{this}(5)$ 
then obtain  $La$  where  $La \in \# D$  and get-level  $La\ (\text{Propagated}\ L\ C'\ \# M) = \text{backtrack-lvl}\ S$ 
  using skip confl-inv by auto
moreover
  have 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: atm-of-eq-atm-of)
    have  $\text{Propagated}\ L\ C'\ \# M \models_{as} CNot\ D$ 
      using conflicting tr-S\ D unfolding cdclW-conflicting-def by auto
    then have  $-L \in \text{lits-of}\ M$ 
      using  $\langle La \in \# D \rangle$  in-CNot-implies-uminus(2)[of  $D\ L\ \text{Propagated}\ L\ C'\ \# M$ ] unfolding  $La$ 
      by auto
    then show False using lev tr-S unfolding cdclW-M-level-inv-def consistent-interp-def by auto
  qed
  then have get-level  $La\ (\text{Propagated}\ L\ C'\ \# M) = \text{get-level}\ La\ M$  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_W-cp-propagate-confl*:
 assumes *cdcl_W-cp* $S\ T$
 shows *propagate*** $S\ T \vee (\exists S'. \text{propagate**}\ S\ S' \wedge \text{conflict}\ S'\ T)$
 using *assms* by *induction blast+*

lemma *rtranclp-cdcl_W-cp-propagate-confl*:
 assumes *cdcl_W-cp*** $S\ T$
 shows *propagate*** $S\ T \vee (\exists S'. \text{propagate**}\ S\ S' \wedge \text{conflict}\ S'\ T)$
 by (*simp add: assms rtranclp-cdcl_W-cp-propa-or-propa-confl*)

lemma *cdcl_W-cp-propagate-completeness*:
 assumes $MN: \text{set}\ M \models_s \text{set-mset}\ N$ and
 cons: *consistent-interp* (*set* M) and
 tot: *total-over-m* (*set* M) (*set-mset* N) and
 lits-of (*trail* S) $\subseteq \text{set}\ M$ and
 init-clss $S = N$ and
 propagate** $S\ S'$ and
 learned-clss $S = \{\#\}$
 shows $\text{length}\ (\text{trail}\ S) \leq \text{length}\ (\text{trail}\ S') \wedge \text{lits-of}\ (\text{trail}\ S') \subseteq \text{set}\ M$
 using *assms*(6,4,5,7)
 proof (*induction rule: rtranclp-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 ⊨s CNot C
  using M'-C LM Y unfolding true-annots-def Ball-def true-annot-def true-clss-def true-clss-def
  by force
moreover
  have set M ⊨ C + {#L#}
  using MN C learned Y unfolding true-clss-def clauses-def
  by (metis NS ⟨init-clss S = init-clss Y⟩ ⟨learned-clss Y = {#}⟩ add.right-neutral
    mem-set-mset-iff)
ultimately have L ∈ 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: set M ⊨s set-mset N
and cons: consistent-interp (set M)
and tot: total-over-m (set M) (set-mset N)
and alien: no-strange-atm S
and learned: learned-clss S = {#}
and clsS[simp]: init-clss S = N
and lits: lits-of (trail S) ⊆ set M
shows ∃ S'. propagate** S S' ∧ full cdclW-cp S S'
proof –
obtain S' where full: full cdclW-cp S S'
  using always-exists-full-cdclW-cp-step alien by blast
then consider (propa) propagate** S S'
  | (confl) ∃ X. propagate** S X ∧ conflict X S'
  using rtrancp-cdclW-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: propagate** S X and
    Xconf: conflict X S'

```



```

by blast
have clsX: init-clss X = init-clss S
  using X by induction auto
have learnedX: learned-clss X = {#} using X learned by induction auto
obtain E where
  E: E ∈# init-clss X + learned-clss X and
  Not-E: trail X ⊨as CNot E
  using Xconf by (auto simp add: conflict.simps clauses-def)
have lits-of (trail X) ⊆ set M
  using cdclW-cp-propagate-completeness[OF assms(1-3) lits - X learned] learned by auto
then have MNE: set M ⊨s CNot E
  using Not-E
  by (fastforce simp add: true-annots-def true-annot-def true-clss-def true-cl-def)
have ¬ set M ⊨s 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

```

See also $cdcl_W\text{-cp}^{**} \text{ ?}S \text{ ?}S' \implies \exists M. \text{trail } ?S' = M @ \text{trail } ?S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$

lemma *rtrancp-propagate-is-trail-append*:
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*:
propagate^{**} S T $\implies cdcl_W\text{-M-level-inv } S \implies T \sim \text{delete-trail-and-rebuild } (\text{trail } T) S$

proof (*induction rule: rtrancp-induct*)

```

case base
then show ?case unfolding state-eq-def by (auto simp: cdcl_W-M-level-inv-decomp)
next
case (step T U) note IH=this(3)[OF this(4)]
moreover have cdclW-M-level-inv U
  using rtrancp-cdclW-consistent-inv ⟨propagate** S T⟩ ⟨propagate T U⟩
  rtrancp-mono[of propagate cdclW] cdclW-cp-consistent-inv propagate'
  rtrancp-propagate-is-rtrancp-cdclW step.prems by blast
then have no-dup (trail U) unfolding cdclW-M-level-inv-def by auto
ultimately show ?case using ⟨propagate T U⟩ unfolding state-eq-def by fastforce
qed

```

lemma *cdcl_W-stgy-strong-completeness-n*:

assumes

MN: *set* M ⊨_s *set-mset* N **and**
 cons: *consistent-interp* (*set* M) **and**
 tot: *total-over-m* (*set* M) (*set-mset* N) **and**
 atm-incl: *atm-of* ' (*set* M) ⊆ *atms-of-msu* 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' ⊆ *set* M \wedge
 no-dup M' \wedge
 $S \sim \text{update-backtrack-lvl } k (\text{append-trail } (\text{rev } M') (\text{init-state } N)) \wedge$
cdcl_W-stgy^{**} (*init-state* N) S

using *length*

```

proof (induction n)
  case 0
  have update-backtrack-lvl 0 (append-trail (rev []) (init-state N)) ~ init-state N
    by (auto simp: state-eq-def simp del: state-simp)
  moreover have
    0 ≤ length [] and
    lits-of [] ⊆ set M and
    cdclW-stgy** (init-state N) (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 = this(1) and n = this(2)
then obtain M' k S where
  l-M': length M' ≥ n and
  M': lits-of M' ⊆ set M and
  n-d[simp]: no-dup M' and
  S: S ~ update-backtrack-lvl k (append-trail (rev M') (init-state N)) and
  st: cdclW-stgy** (init-state N) S
  by auto
have
  M: cdclW-M-level-inv S and
  alien: no-strange-atm S
    using rtrancpl-cdclW-consistent-inv[OF rtrancpl-cdclW-stgy-rtrancpl-cdclW[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: ¬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) ≤ length (trail S') ∧ lits-of (trail S') ⊆ 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  $\langle \text{propagate}^{++} S S' \rangle$ ] S
    rtrancpl-propagate-is-update-trail[of S S'] S M unfolding state-eq-def by simp-all
have S-S':  $S' \sim \text{update-backtrack-lvl} (\text{backtrack-lvl } S')$ 
    (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  $\langle \text{init-clss } S' = N \rangle$  apply simp
    using  $\langle \text{cdcl}_W\text{-stgy } S S' \rangle$  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-confli: no-step conflict S
          proof –
            { fix D
              assume  $D \in \# N$  and  $M' \models_{as} CNot D$ 
              then have  $\text{set } M \models D$  using MN unfolding true-clss-def by auto
              moreover have  $\text{set } M \models_s CNot D$ 
                using  $\langle M' \models_{as} CNot D \rangle M'$ 
                by (metis le-iff-sup true-annots-true-clss 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 cdclW-M-level-inv-def by auto
          then have  $\text{card (lits-of } M') = \text{length } M'$ 
            by (induction M') (auto simp add: lits-of-def card-insert-if)
          then have  $\text{lits-of } M' \subset \text{set } M$ 
            using  $n M' n' \text{ lenM}$  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-msu (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  $\text{lits-of (trail ?S')} \subseteq \text{set } M$  using  $m M' S \text{ undef}$  by auto
            moreover have no-strange-atm ?S'

```

```

    using alien dec M by (meson cdclW-no-strange-atm-inv decide other)
  ultimately 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'] S undef by auto
  have cdclW-M-level-inv ?S'
    using M dec rtrancp-mono[of decide cdclW] by (meson cdclW-consistent-inv decide other)
  then have lev'': cdclW-M-level-inv S''
    using S'' rtrancp-cdclW-consistent-inv rtrancp-propagate-is-rtrancp-cdclW by blast
  then have n-d'': no-dup (trail S'')
    unfolding cdclW-M-level-inv-def by auto
  have length (trail ?S') ≤ length (trail S'') ∧ lits-of (trail S'') ⊆ set M
    using S'' full cdclW-cp-propagate-completeness[OF assms(1-3), of ?S' S''] m M' S undef
    by simp
  then have Suc n ≤ length (trail S'') ∧ lits-of (trail S'') ⊆ set M
    using l-M' S undef by auto
  moreover
    have cdclW-M-level-inv (cons-trail (Marked m (Suc (backtrack-lvl S)))
      (update-backtrack-lvl (Suc (backtrack-lvl S)) S))
      using S (cdclW-M-level-inv (cons-trail (Marked m (k + 1)) (incr-lvl S))) by auto
    then have S'': S'' ∼ update-backtrack-lvl (backtrack-lvl S'')
      (append-trail (rev (trail S'')) (init-state N))
      using rtrancp-propagate-is-update-trail[OF S''] S undef n-d'' lev''
      by (auto simp del: state-simp simp: state-eq-def )
    then have cdclW-stgy** (init-state N) S''
      using cdclW-stgy.intros(2)[OF decide[OF dec] - full] no-step no-confl st
      by (auto simp: cdclW-cp.simps)
    ultimately show ?thesis using S'' n-d'' by blast
  qed
}
ultimately show ?case by blast
qed

```

lemma *cdcl_W-stgy-strong-completeness:*

```

  assumes MN: set M ⊨s set-mset N
  and cons: consistent-interp (set M)
  and tot: total-over-m (set M) (set-mset N)
  and atm-incl: atm-of ' (set M) ⊆ atms-of-msu 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 rtrncpl-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-confl* ($S::'st$) \equiv
 $(\forall M K i M' D. M' @ \text{Marked } K i \# M = \text{trail } S \longrightarrow D \in \# \text{ clauses } S$
 $\longrightarrow \neg M \models_{as} \text{CNot } D)$

lemma *no-smaller-confl-init-sate*[simp]:
no-smaller-confl (init-state N) **unfolding** *no-smaller-confl-def* **by** auto

lemma *cdcl_W-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  $\neg M' \models_{as} CNot\ Da$  using no-f D confl T by auto
  }
  ultimately show  $\neg M' \models_{as} CNot\ Da$  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 ∈ # clauses S
    then have  $\neg M \models_{as} CNot\ Da$  using (tl M' @ Marked K' ia # M = M1) M confl undef smaller
      unfolding no-smaller-confl-def by auto
  }
  moreover {
    assume Da: Da = D + {#L#}
    have  $\neg M \models_{as} CNot\ Da$ 
    proof (rule ccontr)
      assume  $\neg ?thesis$ 
      then have  $-L \in lits\text{-}of\ M$  unfolding Da by auto
      then have  $-L \in lits\text{-}of\ (Propagated\ L\ ((D + \{ \#L\# \})) \# M1)$ 
        using UnI2 (tl M' @ Marked K' ia # M = M1)
        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  $T\ undef\ \langle Da = D + \{\#L\#\} \implies \neg M \models_{as} CNot\ Da \rangle\ decomp\ lev$ 
      unfolding  $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$  by fastforce
  qed
qed

lemma conflict-no-smaller-conf-inv:
  assumes conflict  $S\ S'$ 
  and no-smaller-conf  $S$ 
  shows no-smaller-conf  $S'$ 
  using assms unfolding no-smaller-conf-def by fastforce

lemma propagate-no-smaller-conf-inv:
  assumes propagate: propagate  $S\ S'$ 
  and n-l: no-smaller-conf  $S$ 
  shows no-smaller-conf  $S'$ 
  unfolding no-smaller-conf-def
proof (intro allI impI)
  fix  $M' K i M'' D$ 
  assume  $M': M'' @ Marked\ K\ i\ \# M' = trail\ S'$ 
  and  $D \in \# clauses\ S'$ 
  obtain  $M\ N\ U\ k\ C\ L$  where
     $S: state\ S = (M, N, U, k, C\text{-}True)$  and
     $S': state\ S' = (Propagated\ L\ ((C + \{\#L\#\}))\ \# M, N, U, k, C\text{-}True)$  and
     $C + \{\#L\#\} \in \# clauses\ S$  and
     $M \models_{as} CNot\ C$  and
    undefined-lit  $M\ L$ 
  using propagate by auto
  have  $tl\ M'' @ Marked\ K\ i\ \# M' = trail\ S$  using  $M'\ S\ S'$ 
    by (metis Pair-inject list.inject list.sel(3) marked-lit.distinct(1) self-append-conv2
      tl-append2)
  then have  $\neg M' \models_{as} CNot\ D$ 
    using  $\langle D \in \# clauses\ S' \rangle\ n\text{-}l\ S\ S'\ clauses\text{-}def$  unfolding no-smaller-conf-def by auto
  then show  $\neg M' \models_{as} CNot\ D$  by auto
qed

lemma cdcl_W-cp-no-smaller-conf-inv:
  assumes propagate:  $cdcl_W\text{-}cp\ S\ S'$ 
  and n-l: no-smaller-conf  $S$ 
  shows no-smaller-conf  $S'$ 
  using assms
proof (induct rule:  $cdcl_W\text{-}cp.induct$ )
  case (conflict'  $S\ S'$ )
  then show ?case using conflict-no-smaller-conf-inv[of  $S\ S'$ ] by blast
next
  case (propagate'  $S\ S'$ )
  then show ?case using propagate-no-smaller-conf-inv[of  $S\ S'$ ] by fastforce
qed

lemma rtrancp-cdcl_W-cp-no-smaller-conf-inv:
  assumes propagate:  $cdcl_W\text{-}cp^{**}\ S\ S'$ 
  and n-l: no-smaller-conf  $S$ 
  shows no-smaller-conf  $S'$ 
  using assms

```

```

proof (induct rule: rtrancp-induct)
  case base
  then show ?case by simp
next
  case (step  $S' S''$ )
  then show ?case using cdclW-cp-no-smaller-conflict-inv[of  $S' S''$ ] by fast
qed

lemma trancp-cdclW-cp-no-smaller-conflict-inv:
  assumes propagate: cdclW-cp++  $S S'$ 
  and n-l: no-smaller-conflict  $S$ 
  shows no-smaller-conflict  $S'$ 
  using assms
proof (induct rule: trancp.induct)
  case (r-into-tranc  $S S'$ )
  then show ?case using cdclW-cp-no-smaller-conflict-inv[of  $S S'$ ] by blast
next
  case (tranc-into-tranc  $S S' S''$ )
  then show ?case using cdclW-cp-no-smaller-conflict-inv[of  $S' S''$ ] by fast
qed

lemma full-cdclW-cp-no-smaller-conflict-inv:
  assumes full cdclW-cp  $S S'$ 
  and n-l: no-smaller-conflict  $S$ 
  shows no-smaller-conflict  $S'$ 
  using assms unfolding full-def
  using rtrancp-cdclW-cp-no-smaller-conflict-inv[of  $S S'$ ] by blast

lemma full1-cdclW-cp-no-smaller-conflict-inv:
  assumes full1 cdclW-cp  $S S'$ 
  and n-l: no-smaller-conflict  $S$ 
  shows no-smaller-conflict  $S'$ 
  using assms unfolding full1-def
  using trancp-cdclW-cp-no-smaller-conflict-inv[of  $S S'$ ] by blast

lemma cdclW-stgy-no-smaller-conflict-inv:
  assumes cdclW-stgy  $S S'$ 
  and n-l: no-smaller-conflict  $S$ 
  and conflict-is-false-with-level  $S$ 
  and cdclW-M-level-inv  $S$ 
  shows no-smaller-conflict  $S'$ 
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict'  $S'$ )
  then show ?case using full1-cdclW-cp-no-smaller-conflict-inv[of  $S S'$ ] by blast
next
  case (other'  $S' S''$ )
  have no-smaller-conflict  $S'$ 
    using cdclW-o-no-smaller-conflict-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-cdclW-cp-no-smaller-conflict-inv[of  $S' S''$ ] other'.hyps by blast
qed

lemma conflict-conflict-is-no-clause-is-false-test:

```


assumes *conflict S S'*
and $(\forall D \in \# \text{ init-clss } S + \text{ learned-clss } S. \text{ trail } S \models_{as} CNot D$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{ get-level } L (\text{ trail } S) = \text{ backtrack-lvl } S))$
shows $\forall D \in \# \text{ init-clss } S' + \text{ learned-clss } S'. \text{ trail } S' \models_{as} CNot D$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{ get-level } L (\text{ trail } S') = \text{ backtrack-lvl } S')$
using *assms by auto*

lemma *is-conflicting-exists-conflict:*

assumes $\neg(\forall D \in \# \text{ init-clss } S' + \text{ learned-clss } S'. \neg \text{ trail } S' \models_{as} CNot D)$
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_W-o-conflict-is-no-clause-is-false:*

fixes *S S' :: 'st*
assumes
cdcl_W-o S S' and
lev: cdcl_W-M-level-inv S and
max-lev: conflict-is-false-with-level S and
no-f: no-clause-is-false S and
no-l: no-smaller-conf S
shows *no-clause-is-false S'*
 $\vee (\text{ conflicting } S' = C\text{-True}$
 $\longrightarrow (\forall D \in \# \text{ clauses } S'. \text{ trail } S' \models_{as} CNot D$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{ get-level } L (\text{ trail } S') = \text{ backtrack-lvl } S')))$
using *assms(1,2)*

proof (*induct rule: cdcl_W-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 $\neg ?M \models_{as} CNot D$

using *no-f D S T undef by auto*

have $-L \in \# D$

proof (*rule ccontr*)

assume $\neg ?thesis$

have $?M \models_{as} CNot D$

unfolding *true-annots-def Ball-def true-annot-def CNot-def true-cls-def*

proof (*intro allI impI*)

fix *x*

assume $x: x \in \{\{\# - L\# \} \mid L. L \in \# D\}$

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 $\exists L \in \# x. \text{ lits-of } ?M \models_l L$ **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  $\langle \neg ?M \models_{as} CNot\ D \rangle$  by auto
  qed
have atm-of  $L \notin \text{atm-of } \langle \text{lits-of } ?M \rangle$ 
  using undef defined-lit-map unfolding lits-of-def by fastforce
then have get-level  $(-L) (\text{Marked } L (?k + 1) \# ?M) = ?k + 1$  by simp
then show  $\exists La. La \in \# D \wedge \text{get-level } La\ ?M'$ 
  = backtrack-lvl  $T$ 
  using  $\langle -L \in \# D \rangle\ T$  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  $T = \text{this}(7)$ 
show  $?case$ 
proof (rule HOL.disjI2, clarify)
  fix  $Da$ 
  assume  $Da: Da \in \# \text{clauses } T$ 
  and  $M-D: \text{trail } T \models_{as} CNot\ Da$ 
  obtain  $c$  where  $M: \text{trail } S = c @ M2 @ \text{Marked } K\ (i + 1) \# M1$ 
  using decomp by auto
  have  $tr-T: \text{trail } T = \text{Propagated } L\ (D + \{\#L\}) \# M1$ 
  using  $T\ \text{decomp}\ \text{undef}\ \text{lev}$  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  $lev': \text{cdcl}_W\text{-M-level-inv } T$ 
  using cdclW-consistent-inv lev other by blast
  then have  $-L \notin \text{lits-of } M1$ 
  unfolding cdclW-M-level-inv-def lits-of-def
  proof -
    have consistent-interp  $(\text{lits-of } (\text{trail } S)) \wedge \text{no-dup } (\text{trail } S)$ 
       $\wedge \text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ 
       $\wedge \text{get-all-levels-of-marked } (\text{trail } S)$ 
      =  $\text{rev } [1..<1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$ 
    using  $lev\ \text{cdcl}_W\text{-M-level-inv-def}$  by blast
    then show  $-L \notin \text{lit-of } \langle \text{set } M1 \rangle$ 
      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  $Da \in \# \text{clauses } S$ 
  then have  $\neg M1 \models_{as} CNot\ Da$  using no-l M unfolding no-smaller-confl-def by auto
}
moreover {
  assume  $Da: Da = D + \{\#L\}$ 
  have  $\neg M1 \models_{as} CNot\ Da$  using  $\langle -L \notin \text{lits-of } M1 \rangle$  unfolding  $Da$  by simp
}
ultimately have  $\neg M1 \models_{as} CNot\ Da$ 
  using  $Da\ T\ \text{undef}\ \text{decomp}\ \text{lev}$  by (fastforce simp: cdclW-M-level-inv-decomp)
then have  $-L \in \# Da$ 
  using  $M-D\ \langle -L \notin \text{lits-of } M1 \rangle$  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-cdcl_W-cp-exists-conflict-decompose:

```

  assumes confl: ∃ D ∈ #clauses S. trail S ⊨as CNot D
  and full: full cdclW-cp S U
  and no-confl: 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-confl)
  then show ?thesis
  proof cases
    case confl
    then show ?thesis by blast
  next
    case propa
    then have conflicting U = C-True
      using no-confl by induction auto
    moreover have [simp]: learned-clss U = learned-clss S and
      [simp]: init-clss U = init-clss S
      using propa by induction auto
    moreover
      obtain D where D: D ∈ #clauses U and
        trS: trail S ⊨as CNot D
      using confl clauses-def by auto
    obtain M where M: trail U = M @ trail S
      using full rtranclp-cdclW-cp-dropWhile-trail unfolding full-def by meson
    have tr-U: trail U ⊨as CNot D
      apply (rule true-annots-mono)
      using trS unfolding M by simp-all
    have ∃ V. conflict U V
      using (conflicting U = C-True) D clauses-def not-conflict-not-any-negated-init-clss 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_W-cp-exists-conflict-full1-decompose:

```

  assumes confl: ∃ D ∈ #clauses S. trail S ⊨as CNot D
  and full: full cdclW-cp S U

```

and *no-conflict*: *conflicting* $S = C\text{-True}$
shows $\exists T D. \text{propagate}^{**} S T \wedge \text{conflict} T U$
 $\wedge \text{trail } T \models_{as} C\text{Not } D \wedge \text{conflicting } U = C\text{-Clause } D \wedge D \in \# \text{ clauses } S$
proof –
obtain T **where** *propa*: $\text{propagate}^{**} S T$ **and** *conf*: $\text{conflict} T U$
using *full1-cdcl_W-cp-exists-conflict-decompose*[*OF* *assms*] **by** *blast*
have p : *learned-clss* $T = \text{learned-clss } S \text{ init-clss } T = \text{init-clss } S$
using *propa* **by** *induction auto*
have c : *learned-clss* $U = \text{learned-clss } T \text{ init-clss } U = \text{init-clss } T$
using *conf* **by** *induction auto*
obtain D **where** $\text{trail } T \models_{as} C\text{Not } D \wedge \text{conflicting } U = C\text{-Clause } D \wedge D \in \# \text{ clauses } S$
using *conf* p c **by** (*fastforce simp: clauses-def*)
then show *?thesis*
using *propa conf* **by** *blast*
qed

lemma *cdcl_W-stgy-no-smaller-conflict*:
assumes *cdcl_W-stgy* $S S'$
and *n-l*: *no-smaller-conflict* S
and *conflict-is-false-with-level* S
and *cdcl_W-M-level-inv* S
and *no-clause-is-false* S
and *distinct-cdcl_W-state* S
and *cdcl_W-conflicting* S
shows *no-smaller-conflict* S'
using *assms*
proof (*induct rule: cdcl_W-stgy.induct*)
case (*conflict'* S')
show *no-smaller-conflict* S'
using *conflict'.hyps conflict'.prems(1) full1-cdcl_W-cp-no-smaller-conflict-inv* **by** *blast*
next
case (*other'* $S' S''$)
have lev' : *cdcl_W-M-level-inv* S'
using *cdcl_W-consistent-inv other other'.hyps(1) other'.prems(3)* **by** *blast*
show *no-smaller-conflict* S''
using *cdcl_W-stgy-no-smaller-conflict-inv*[*OF cdcl_W-stgy.other'*[*OF other'.hyps(1-3)*]]
 $\text{other}'.\text{prems}(1-3)$ **by** *blast*
qed

lemma *cdcl_W-stgy-ex-lit-of-max-level*:
assumes *cdcl_W-stgy* $S S'$
and *n-l*: *no-smaller-conflict* S
and *conflict-is-false-with-level* S
and *cdcl_W-M-level-inv* S
and *no-clause-is-false* S
and *distinct-cdcl_W-state* S
and *cdcl_W-conflicting* S
shows *conflict-is-false-with-level* S'
using *assms*
proof (*induct rule: cdcl_W-stgy.induct*)
case (*conflict'* S')
have *no-smaller-conflict* S'
using *conflict'.hyps conflict'.prems(1) full1-cdcl_W-cp-no-smaller-conflict-inv* **by** *blast*
moreover have *conflict-is-false-with-level* S'
using *conflict'.hyps conflict'.prems(2-4)*

```

  rtrancp-cdclW-co-conflict-ex-lit-of-max-level[of  $S$   $S'$ ]
  unfolding full-def full1-def rtrancp-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  $S'$ 
     $\vee$  (conflicting  $S' = C\text{-True} \longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{as} C\text{Not } D$ 
       $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S'))$ )
    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  $S' = C\text{-True}$ 
    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 rtrancp-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: rtrancp-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 <conflicting  $S' = C\text{-True}$ > 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 - - ⟨conflicting S' = C-True⟩]
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 @ trail S' and nm: ∀ m ∈ set M. ¬is-marked m
using rtrancp-cdclW-cp-dropWhile-trail other'(3) unfolding full-def by meson
have btS: backtrack-lvl S'' = backtrack-lvl S'
using other'.hyps(3) unfolding full-def by (metis rtrancp-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' ⊨as CNot D
moreover then obtain L where L ∈ # D and get-level L (trail S') = backtrack-lvl S'
using D-L D by blast
moreover
have LS': -L ∈ lits-of (trail S')
using ⟨trail S' ⊨as CNot D⟩ ⟨L ∈ # D⟩ in-CNot-implies-uminus(2) by blast
{ fix x :: ('v, nat, 'v literal multiset) marked-lit and
xb :: ('v, nat, 'v literal multiset) marked-lit
assume a1: x ∈ set (trail S') and
a2: xb ∈ 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-annots-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  $LS'$ :  $\text{atm-of } L \notin \text{atm-of ' lits-of (trail } S')$ 
    using  $nd \langle L \in \# D \rangle LM$  unfolding  $M$  by ( $\text{auto simp add: lits-of-def}$ )
  show ?thesis
  proof cases
    assume  $ne$ :  $\text{get-all-levels-of-marked (trail } S') = []$ 
    have  $\text{backtrack-lvl } S'' = 0$ 
      using  $inv$   $ne$   $nm$  unfolding  $\text{cdcl}_W\text{-}M\text{-level-inv-def } M$ 
      by ( $\text{simp add: get-all-levels-of-marked-nil-iff-not-is-marked}$ )
    moreover
      have  $a1$ :  $\text{get-rev-level } L \ 0 \ (\text{rev } M) = 0$ 
        using  $nm$  by auto
      then have  $\text{get-level } L \ (M @ \text{trail } S') = 0$ 
        by ( $\text{metis } LS' \text{ get-all-levels-of-marked-nil-iff-not-is-marked}$ 
           $\text{get-level-skip-beginning-not-marked lits-of-def } ne$ )
    ultimately show ?thesis using  $\langle \text{conflicting } S'' = C\text{-Clause } D \rangle \langle L \in \# D \rangle$  unfolding  $M$ 
      by auto
  next
    assume  $ne$ :  $\text{get-all-levels-of-marked (trail } S') \neq []$ 
    have  $hd$  ( $\text{get-all-levels-of-marked (trail } S')$ ) =  $\text{backtrack-lvl } S'$ 
      using  $ne$   $lev' M$   $nm$  unfolding  $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ 
      by ( $\text{cases get-all-levels-of-marked (trail } S')$ )
      ( $\text{simp-all add: get-all-levels-of-marked-nil-iff-not-is-marked[symmetric]}$ )
    moreover have  $\text{atm-of } L \in \text{atm-of ' lits-of } M$ 
      using  $\langle -L \in \text{lits-of } M \rangle$ 
      by ( $\text{simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def}$ )
    ultimately show ?thesis
      using  $nm$   $ne$   $\langle L \in \# D \rangle \langle \text{conflicting } S'' = C\text{-Clause } D \rangle$ 
       $\text{get-level-skip-beginning-hd-get-all-levels-of-marked}[OF \ LS', \text{ of } M]$ 
       $\text{get-level-skip-in-all-not-marked}[of \text{ rev } M \ L \ \text{backtrack-lvl } S']$ 
      unfolding  $\text{lits-of-def } btS \ M$ 
      by auto
    qed
  qed
}
ultimately have  $\text{conflict-is-false-with-level } S''$  by blast
}
moreover
{
  assume  $\text{conflicting } S' \neq C\text{-True}$ 
  have  $\text{no-clause-is-false } S'$  using  $\langle \text{conflicting } S' \neq C\text{-True} \rangle$  by auto
  then have  $\text{conflict-is-false-with-level } S''$  using  $\text{calculation}(\mathcal{I})$  by blast
}
ultimately show ?case by fast
qed

```

lemma $\text{rtrancp-cdcl}_W\text{-stgy-no-smaller-confl-inv}$:

assumes
 $\text{cdcl}_W\text{-stgy}^{**} \ S \ S'$ **and**
 $n\text{-l: no-smaller-confl } S$ **and**
 $\text{cls-false: conflict-is-false-with-level } S$ **and**
 $\text{lev: cdcl}_W\text{-}M\text{-level-inv } S$ **and**
 $\text{no-f: no-clause-is-false } S$ **and**
 $\text{dist: distinct-cdcl}_W\text{-state } S$ **and**
 $\text{conflicting: cdcl}_W\text{-conflicting } S$ **and**

decomp: *all-decomposition-implies-m* (*init-clss* *S*) (*get-all-marked-decomposition* (*trail* *S*)) **and**
learned: *cdcl_W-learned-clause* *S* **and**
alien: *no-strange-atm* *S*
shows *no-smaller-conf* *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-conf* *S'* **and** *conflict-is-false-with-level* *S'*
using *IH* **by** *blast*+
moreover have *cdcl_W-M-level-inv* *S'*
using *st lev rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*
by (*blast intro*: *rtranclp-cdcl_W-consistent-inv*) +
moreover have *no-clause-is-false* *S'*
using *st no-f rtranclp-cdcl_W-stgy-not-non-negated-init-clss* **by** *blast*
moreover have *distinct-cdcl_W-state* *S'*
using *rtanclp-distinct-cdcl_W-state-inv*[*of S S'*] *lev rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*[*OF st*]
dist **by** *auto*
moreover have *cdcl_W-conflicting* *S'*
using *rtranclp-cdcl_W-all-inv*(6)[*of S S'*] *st alien conflicting decomp dist learned lev*
rtranclp-cdcl_W-stgy-rtranclp-cdcl_W **by** *blast*
ultimately show ?*case*
using *cdcl_W-stgy-no-smaller-conf*[*OF cdcl*] *cdcl_W-stgy-ex-lit-of-max-level*[*OF cdcl*] **by** *fast*
qed

17.6.7 Final States are Conclusive

lemma *full-cdcl_W-stgy-final-state-conclusive-non-false*:
fixes *S' :: 'st*
assumes *full*: *full cdcl_W-stgy* (*init-state* *N*) *S'*
and *no-d*: *distinct-mset-mset* *N*
and *no-empty*: $\forall D \in \#N. D \neq \{\#\}$
shows (*conflicting* *S'* = *C-Clause* $\{\#\}$ \wedge *unsatisfiable* (*set-mset* (*init-clss* *S'*)))
 \vee (*conflicting* *S'* = *C-True* \wedge *trail* *S'* \models_{asm} *init-clss* *S'*)
proof –
let ?*S* = *init-state* *N*
have
termi: $\forall S''. \neg \text{cdcl}_W\text{-stgy } S' S''$ **and**
step: *cdcl_W-stgy** (*init-state* *N*) *S'* **using** *full unfolding full-def* **by** *auto*
moreover have
learned: *cdcl_W-learned-clause* *S'* **and**
level-inv: *cdcl_W-M-level-inv* *S'* **and**
alien: *no-strange-atm* *S'* **and**
no-dup: *distinct-cdcl_W-state* *S'* **and**
conf: *cdcl_W-conflicting* *S'* **and**
decomp: *all-decomposition-implies-m* (*init-clss* *S'*) (*get-all-marked-decomposition* (*trail* *S'*))
using *no-d tranclp-cdcl_W-stgy-tranclp-cdcl_W*[*of ?S S'*] *step rtranclp-cdcl_W-all-inv*(1–6)[*of ?S S'*]
unfolding *rtranclp-unfold* **by** *auto*
moreover
have $\forall D \in \#N. \neg \square \models_{as} CNot D$ **using** *no-empty* **by** *auto*
then have *conf*-*k*: *conflict-is-false-with-level* *S'*
using *rtranclp-cdcl_W-stgy-no-smaller-conf*-*inv*[*OF step*] *no-d* **by** *auto*
show ?*thesis*
using *cdcl_W-stgy-final-state-conclusive*[*OF termi decomp learned level-inv alien no-dup conf*]

$\text{confl-}k]$.
qed

lemma *conflict-is-full1-cdcl_W-cp*:

assumes *cp*: *conflict S S'*

shows *full1 cdcl_W-cp S S'*

proof –

have *cdcl_W-cp S S'* **and** *conflicting S' ≠ C-True* **using** *cp cdcl_W-cp.intros* **by** *auto*

then have *cdcl_W-cp⁺⁺ S S'* **by** *blast*

moreover have *no-step cdcl_W-cp S'*

using $\langle \text{conflicting } S' \neq C\text{-True} \rangle$ **by** $(\text{metis } \text{cdcl}_W\text{-cp-conflicting-not-empty}$
conflicting-clause.exhaust)

ultimately show *full1 cdcl_W-cp S S'* **unfolding** *full1-def* **by** *blast+*

qed

lemma *cdcl_W-cp-fst-empty-conflicting-false*:

assumes *cdcl_W-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_W-o-fst-empty-conflicting-false*:

assumes *cdcl_W-o S S'*

and *trail S = []*

and *conflicting S ≠ C-True*

shows *False*

using *assms* **by** $(\text{induct rule: } \text{cdcl}_W\text{-o.induct})$ *auto*

lemma *cdcl_W-stgy-fst-empty-conflicting-false*:

assumes *cdcl_W-stgy S S'*

and *trail S = []*

and *conflicting S ≠ C-True*

shows *False*

using *assms* **apply** $(\text{induct rule: } \text{cdcl}_W\text{-stgy.induct})$

using *trancpD cdcl_W-cp-fst-empty-conflicting-false* **unfolding** *full1-def* **apply** *metis*

using *cdcl_W-o-fst-empty-conflicting-false* **by** *blast*

thm *cdcl_W-cp.induct[split-format(complete)]*

lemma *cdcl_W-cp-conflicting-is-false*:

cdcl_W-cp S S' ⇒ conflicting S = C-Clause {#} ⇒ False

by $(\text{induction rule: } \text{cdcl}_W\text{-cp.induct})$ *auto*

lemma *rtrancp-cdcl_W-cp-conflicting-is-false*:

cdcl_W-cp⁺⁺ S S' ⇒ conflicting S = C-Clause {#} ⇒ False

apply $(\text{induction rule: } \text{trancp.induct})$

by $(\text{auto dest: } \text{cdcl}_W\text{-cp-conflicting-is-false})$

lemma *cdcl_W-o-conflicting-is-false*:

cdcl_W-o S S' ⇒ conflicting S = C-Clause {#} ⇒ False

by $(\text{induction rule: } \text{cdcl}_W\text{-o.induct})$ *auto*

lemma *cdcl_W-stgy-conflicting-is-false*:

$cdcl_W\text{-stgy } S S' \implies \text{conflicting } S = C\text{-Clause } \{\#\} \implies \text{False}$
apply (induction rule: $cdcl_W\text{-stgy.induct}$)
unfolding $full1\text{-def}$ **apply** (metis (no-types) $cdcl_W\text{-cp-conflicting-not-empty trancpD}$)
unfolding $full\text{-def}$ **by** (metis $conflict\text{-with-false-implies-terminated other}$)

lemma $rtrancp\text{-}cdcl_W\text{-stgy-conflicting-is-false}$:
 $cdcl_W\text{-stgy}^{**} S S' \implies \text{conflicting } S = C\text{-Clause } \{\#\} \implies S' = S$
apply (induction rule: $rtrancp\text{-induct}$)
apply $simp$
using $cdcl_W\text{-stgy-conflicting-is-false}$ **by** $blast$

lemma $full\text{-}cdcl_W\text{-init-clss-with-false-normal-form}$:
assumes
 $\forall m \in \text{set } M. \neg \text{is-marked } m$ **and**
 $E = C\text{-Clause } D$ **and**
 $\text{state } S = (M, N, U, 0, E)$
 $full\ cdcl_W\text{-stgy } S S'$ **and**
 $\text{all-decomposition-implies-}m\ (\text{init-clss } S)\ (\text{get-all-marked-decomposition } (\text{trail } S))$
 $cdcl_W\text{-learned-clause } S$
 $cdcl_W\text{-}M\text{-level-inv } S$
 $\text{no-strange-atm } S$
 $\text{distinct-}cdcl_W\text{-state } S$
 $cdcl_W\text{-conflicting } S$
shows $\exists M''. \text{state } S' = (M'', N, U, 0, C\text{-Clause } \{\#\})$
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 $rtrancp\text{-}cdcl_W\text{-stgy-conflicting-is-false}$ **unfolding** $full\text{-def } cdcl_W\text{-conflicting-def}$ **by** $auto$
next
case $(\text{Cons } L M)$ **note** $IH = \text{this}(1)$ **and** $full = \text{this}(8)$ **and** $E = \text{this}(10)$ **and** $inv = \text{this}(2-7)$ **and**
 $S = \text{this}(9)$ **and** $nm = \text{this}(11)$
obtain $K p$ **where** $K: L = \text{Propagated } K p$
using nm **by** (cases L) $auto$
have $\text{every-mark-is-a-conflict } S$ **using** inv **unfolding** $cdcl_W\text{-conflicting-def}$ **by** $auto$
then have $MpK: M \models_{as} CNot\ (p - \{\#K\# \})$ **and** $Kp: K \in \# p$
using S **unfolding** K **by** $fastforce+$
then have $p: p = (p - \{\#K\# \}) + \{\#K\# \}$
by ($auto\ simp\ add: \text{multiset-eq-iff}$)
then have $K': L = \text{Propagated } K\ ((p - \{\#K\# \}) + \{\#K\# \})$
using K **by** $auto$

consider $(D) D = \{\#\} \mid (D') D \neq \{\#\}$ **by** $blast$
then show $?case$
proof $cases$
case D
then show $?thesis$
using $full\ rtrancp\text{-}cdcl_W\text{-stgy-conflicting-is-false } S$ **unfolding** $full\text{-def } E D$ **by** $auto$
next
case D'
then have $\text{no-p: no-step propagate } S$ **and** $\text{no-c: no-step conflict } S$
using $S E$ **by** $auto$
then have $\text{no-step } cdcl_W\text{-cp } S$ **by** ($auto\ simp: cdcl_W\text{-cp.simps}$)
have $\text{res-skip: } \exists T. (\text{resolve } S T \wedge \text{no-step skip } S \wedge \text{full } cdcl_W\text{-cp } T T)$
 $\vee (\text{skip } S T \wedge \text{no-step resolve } S \wedge \text{full } cdcl_W\text{-cp } T T)$

```

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  $full\ cdcl_W\text{-}cp\ T\ T$ 
    using  $sk$  by (auto simp add: conflicting-clause-full-cdclW-cp)
  then show ?thesis
    using  $sk\ res$  by blast
next
  assume  $LD: \neg \neg \text{lit-of } L \notin \# D$ 
  then have  $D$ : C-Clause  $D = C\text{-Clause } ((D - \{\# \neg \text{lit-of } L \#\}) + \{\# \neg \text{lit-of } L \#\})$ 
    by (auto simp add: multiset-eq-iff)

  have  $\bigwedge L. \text{get-level } L\ M = 0$ 
    by (simp add: nm)
  then have  $\text{get-maximum-level } (D - \{\# \neg K \#\})$ 
    (Propagated  $K\ ((p - \{\# K \#\} + \{\# K \#\})) \# M = 0$ 
  using  $LD$  get-maximum-level-exists-lit-of-max-level
  proof  $-$ 
    obtain  $L'$  where  $\text{get-level } L' (L \# M) = \text{get-maximum-level } D (L \# M)$ 
      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  $S\ K\ p - \{\# K \#\}\ M\ N\ U\ 0\ (D - \{\# \neg K \#\})$ 
    update-conflicting (C-Clause (remdups-mset ( $D - \{\# \neg K \#\} + (p - \{\# K \#\}))))$  (tl-trail  $S$ )]
     $S$  unfolding  $K' D E$  by fastforce
  have  $full\ cdcl_W\text{-}cp\ T\ T$ 
    using  $sk$  by (auto simp add: conflicting-clause-full-cdclW-cp)
  then show ?thesis
    using  $sk\ res$  by blast
  qed
then have  $step\text{-}s: \exists T. \text{cdcl}_W\text{-}stgy\ S\ T$ 
  using (no-step cdclW-cp  $S$ ) other' by (meson bj resolve skip)
have  $\text{get-all-marked-decomposition } (L \# M) = [([], L \# M)]$ 
  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  $no\text{-}b$ : no-step backtrack  $S$ 
  using  $nm\ S$  by auto
have  $no\text{-}d$ : no-step decide  $S$ 
  using  $S\ E$  by auto

have  $full\text{-}S\text{-}S$ :  $full\ cdcl_W\text{-}cp\ S\ S$ 
  using  $S\ E$  by (auto simp add: conflicting-clause-full-cdclW-cp)
then have  $no\text{-}f$ : no-step ( $full1\ cdcl_W\text{-}cp$ )  $S$ 
  unfolding  $full\text{-}def\ full1\text{-}def\ rtranclp\text{-}unfold$  by (meson trancpD)
obtain  $T$  where
   $s: \text{cdcl}_W\text{-}stgy\ S\ T$  and  $st: \text{cdcl}_W\text{-}stgy^{**}\ T\ S'$ 
  using  $full\ step\text{-}s\ full$  unfolding  $full\text{-}def$  by (metis rtranclp-unfold trancpD)
have  $\text{resolve } S\ T \vee \text{skip } S\ T$ 
  using  $s\ no\text{-}b\ no\text{-}d\ res\text{-}skip\ full\text{-}S\text{-}S$  unfolding  $\text{cdcl}_W\text{-}stgy.simps\ \text{cdcl}_W\text{-}o.simps\ full\text{-}unfold$ 
   $full1\text{-}def$ 
  by (auto dest!: trancpD simp: cdclW-bj.simps)
then obtain  $D'$  where  $T$ : state  $T = (M, N, U, 0, C\text{-Clause } D')$ 

```

```

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: {#} ∈ # N
  shows conflicting S' = C-Clause {#} ∧ unsatisfiable (set-mset (init-cls S'))
proof -
  let ?S = init-state N
  have cdclW-stgy** ?S S' and no-step cdclW-stgy S' using full unfolding full-def by auto
  then have plus-or-eq: cdclW-stgy++ ?S S' ∨ S' = ?S unfolding rtrancp-unfold by auto
  have ∃ S''. conflict ?S S'' using empty not-conflict-not-any-negated-init-cls by force

  then have cdclW-stgy: ∃ S'. cdclW-stgy ?S S'
    using cdclW-cp.conflict'[of ?S] conflict-is-full1-cdclW-cp cdclW-stgy.intros(1) by metis
  have S' ≠ ?S using (no-step cdclW-stgy S') cdclW-stgy by blast

  then obtain St:: 'st where St: cdclW-stgy ?S St and cdclW-stgy** St S'
    using plus-or-eq by (metis (no-types) (cdclW-stgy** ?S S') converse-rtrancpE)
  have st: cdclW** ?S St
    by (simp add: rtrancp-unfold (cdclW-stgy ?S St) cdclW-stgy-trancp-cdclW)

  have ∃ T. conflict ?S T
    using empty not-conflict-not-any-negated-init-cls by force
  then have fullSt: full1 cdclW-cp ?S St
    using St unfolding cdclW-stgy.simps by blast
  then have bt: backtrack-lvl St = (0::nat)
    using rtrancp-cdclW-cp-backtrack-lvl unfolding full1-def
    by (fastforce dest!: trancp-into-rtrancp)
  have cls-St: init-cls St = N
    using fullSt cdclW-stgy-no-more-init-cls[OF St] by auto
  have conflicting St ≠ C-True
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ∃ T. conflict St T

```

```

    using empty cls-St by (fastforce simp: clauses-def)
  then show False using fullSt unfolding full1-def by blast
qed

have 1:  $\forall m \in \text{set } (\text{trail } St). \neg \text{is-marked } m$ 
  using fullSt unfolding full1-def by (auto dest!: rtrancplp-into-rtrancplp
    rtrancplp-cdclW-cp-dropWhile-trail)
have 2: full cdclW-stgy St S'
  using  $\langle \text{cdcl}_W\text{-stgy}^{**} St S' \rangle \langle \text{no-step cdcl}_W\text{-stgy } S' \rangle \text{bt}$  unfolding full-def by auto
have 3: all-decomposition-implies-m
  (init-clss St)
  (get-all-marked-decomposition
    (trail St))
  using rtrancplp-cdclW-all-inv(1)[OF st] no-d bt by simp
have 4: cdclW-learned-clause St
  using rtrancplp-cdclW-all-inv(2)[OF st] no-d bt bt by simp
have 5: cdclW-M-level-inv St
  using rtrancplp-cdclW-all-inv(3)[OF st] no-d bt by simp
have 6: no-strange-atm St
  using rtrancplp-cdclW-all-inv(4)[OF st] no-d bt by simp
have 7: distinct-cdclW-state St
  using rtrancplp-cdclW-all-inv(5)[OF st] no-d bt by simp
have 8: cdclW-conflicting St
  using rtrancplp-cdclW-all-inv(6)[OF st] no-d bt by simp
have init-clss S' = init-clss St and conflicting S' = C-Clause {#}
  using  $\langle \text{conflicting } St \neq C\text{-True} \rangle \text{full-cdcl}_W\text{-init-clss-with-false-normal-form}[OF 1, \text{of } - - St]$ 
  2 3 4 5 6 7 8 St apply (metis  $\langle \text{cdcl}_W\text{-stgy}^{**} St S' \rangle \text{rtrancplp-cdcl}_W\text{-stgy-no-more-init-clss}$ )
  using  $\langle \text{conflicting } St \neq C\text{-True} \rangle \text{full-cdcl}_W\text{-init-clss-with-false-normal-form}[OF 1, \text{of } - - St - -$ 
  S'] 2 3 4 5 6 7 8 by (metis bt conflicting-clause.exhaust prod.inject)

moreover have init-clss S' = N
  using  $\langle \text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S' \rangle \text{rtrancplp-cdcl}_W\text{-stgy-no-more-init-clss}$  by fastforce
moreover have unsatisfiable (set-mset N)
  by (meson empty mem-set-mset-iff satisfiable-def true-cls-empty true-clss-def)
ultimately show ?thesis by auto
qed

```

lemma full-cdcl_W-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 {#}  $\wedge$  unsatisfiable (set-mset (init-clss S')))
     $\vee$  (conflicting S' = C-True  $\wedge$  trail S'  $\models_{asm}$  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_W-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 {#}  $\wedge$  unsatisfiable (set-mset N))
     $\vee$  (conflicting S' = C-True  $\wedge$  trail S'  $\models_{asm}$  N  $\wedge$  satisfiable (set-mset N))

```

proof –

```

  have N: init-clss S' = N
    using full unfolding full-def by (auto dest: rtrancplp-cdclW-stgy-no-more-init-clss)

```

```

consider
  (confl) conflicting  $S' = C\text{-Clause } \{\#\}$  and unsatisfiable (set-mset (init-clss  $S'$ ))
  | (sat) conflicting  $S' = C\text{-True}$  and trail  $S' \models_{asm} \text{init-clss } S'$ 
  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_rtrancp-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  $S'$ )  $\models_s$  set-mset (init-clss  $S'$ )
    using sat(2) by (auto simp add: true-annots-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
end
end
theory CDCL-W-Termination
imports CDCL-W
begin

context cdclW-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_W-all-struct-inv* **where**

```

cdclW-all-struct-inv  $S =$ 
  (no-strange-atm  $S \wedge$  cdclW-M-level-inv  $S$ 
 $\wedge (\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s)$ 
 $\wedge \text{distinct-cdcl}_W\text{-state } S \wedge \text{cdcl}_W\text{-conflicting } S$ 
 $\wedge \text{all-decomposition-implies-m (init-clss } S) (\text{get-all-marked-decomposition (trail } S))$ 
 $\wedge \text{cdcl}_W\text{-learned-clause } S)$ 

```

lemma *cdcl_W-all-struct-inv-inv*:

```

assumes cdclW  $S S'$  and cdclW-all-struct-inv  $S$ 
shows cdclW-all-struct-inv  $S'$ 
unfolding cdclW-all-struct-inv-def
proof (intro HOL.conjI)
  show no-strange-atm  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by auto
  show cdclW-M-level-inv  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
  show distinct-cdclW-state  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
  show cdclW-conflicting  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast

```

```

show all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ ))
  using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
show cdclW-learned-clause  $S'$ 
  using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast

show  $\forall s \in \# \text{learned-clss } S'. \neg \text{tautology } s$ 
  using assms(1)[THEN learned-clss-are-not-tautologies] assms(2)
  unfolding cdclW-all-struct-inv-def by fast
qed

```

```

lemma rtranclp-cdclW-all-struct-inv-inv:
  assumes cdclW**  $S$   $S'$  and cdclW-all-struct-inv  $S$ 
  shows cdclW-all-struct-inv  $S'$ 
  using assms by induction (auto intro: cdclW-all-struct-inv-inv)

```

```

lemma cdclW-stgy-cdclW-all-struct-inv:
  cdclW-stgy  $S$   $T \implies \text{cdcl}_W\text{-all-struct-inv } S \implies \text{cdcl}_W\text{-all-struct-inv } T$ 
  by (meson cdclW-stgy-rtranclp-cdclW rtranclp-cdclW-all-struct-inv-inv rtranclp-unfold)

```

```

lemma rtranclp-cdclW-stgy-cdclW-all-struct-inv:
  cdclW-stgy**  $S$   $T \implies \text{cdcl}_W\text{-all-struct-inv } S \implies \text{cdcl}_W\text{-all-struct-inv } T$ 
  by (induction rule: rtranclp-induct) (auto intro: cdclW-stgy-cdclW-all-struct-inv)

```

17.8 No Relearning of a clause

```

lemma cdclW-o-new-clause-learned-is-backtrack-step:
  assumes learned: D  $\in \#$  learned-clss  $T$  and
  new: D  $\notin \#$  learned-clss  $S$  and
  cdclW: cdclW-o  $S$   $T$  and
  lev: cdclW-M-level-inv  $S$ 
  shows backtrack  $S$   $T \wedge \text{conflicting } S = C\text{-Clause } D$ 
  using cdclW lev learned new
proof (induction rule: cdclW-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: cdclW-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 cdclW-cp-new-clause-learned-has-backtrack-step:
  assumes learned: D  $\in \#$  learned-clss  $T$  and
  new: D  $\notin \#$  learned-clss  $S$  and
  cdclW: cdclW-stgy  $S$   $T$  and
  lev: cdclW-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 cdclW learned new
proof (induction rule: cdclW-stgy.induct)
  case (conflict'  $S'$ )
  then show ?case
  unfolding full1-def by (metis (mono-tags, lifting) rtranclp-cdclW-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_W-cp-learned-clause-inv*)
then show ?case
using *cdcl_W-o-new-clause-learned-is-backtrack-step*[*OF - (D ∉ # learned-clss S) (cdcl_W-o S S')*]
(full cdcl_W-cp S' S'') lev by (metis cdcl_W-stgy.conflict' full-unfold r-into-rtranclp
rtranclp.rtrancl-refl)
qed

lemma *rtranclp-cdcl_W-cp-new-clause-learned-has-backtrack-step*:
assumes *learned: D ∈ # learned-clss T and*
new: D ∉ # learned-clss S and
*cdcl_W: cdcl_W-stgy** S T and*
lev: cdcl_W-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_W learned new*
proof (*induction rule: rtranclp-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': cdcl_W-stgy** S S' and*
bt: backtrack S' S'' and
confl: conflicting S' = C-Clause D and
*st'': cdcl_W-stgy** S'' T*
using *IH D-S by metis*
then show ?thesis **using** *o* **by** (*meson rtranclp.simps*)
next
case *False*
have *cdcl_W-M-level-inv T*
using *lev rtranclp-cdcl_W-stgy-consistent-inv st by blast*
then obtain S' **where**
bt: backtrack T S' and
*st': cdcl_W-stgy** S' U and*
confl: conflicting T = C-Clause D
using *cdcl_W-cp-new-clause-learned-has-backtrack-step*[*OF D-U False o*]
by *metis*
then have *cdcl_W-stgy** S T and*
backtrack T S' and
conflicting T = C-Clause D and
*cdcl_W-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 $\text{Marked } K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$
using *assms* **by** *auto*

lemma *conflict-no-more-Marked-lit*:

assumes *conflict* $S S'$
shows $\text{Marked } K i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K i \in \text{set } (\text{trail } S')$
using *assms* **by** *auto*

lemma *cdcl_W-cp-no-more-Marked-lit*:

assumes *cdcl_W-cp* $S S'$
shows $\text{Marked } K i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K i \in \text{set } (\text{trail } S')$
using *assms* **apply** (*induct rule: cdcl_W-cp.induct*)
using *conflict-no-more-Marked-lit* *propagate-no-more-Marked-lit* **by** *auto*

lemma *rtranclp-cdcl_W-cp-no-more-Marked-lit*:

assumes *cdcl_W-cp^{**}* $S S'$
shows $\text{Marked } K i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K i \in \text{set } (\text{trail } S')$
using *assms* **apply** (*induct rule: rtranclp-induct*)
using *cdcl_W-cp-no-more-Marked-lit* **by** *blast+*

lemma *cdcl_W-o-no-more-Marked-lit*:

assumes *cdcl_W-o* $S S'$ **and** *cdcl_W-M-level-inv* S **and** $\neg \text{decide } S S'$
shows $\text{Marked } K i \in \text{set } (\text{trail } S') \longrightarrow \text{Marked } K i \in \text{set } (\text{trail } S)$
using *assms*

proof (*induct rule: cdcl_W-o-induct-lev2*)

case *backtrack* **note** *decomp = this(1)* **and** *undef = this(6)* **and** $T = \text{this}(7)$ **and** $\text{lev} = \text{this}(8)$
then show *?case*
by (*auto simp: cdcl_W-M-level-inv-decomp*)

next

case (*decide* $L T$)
then show *?case* **by** *blast*

qed *auto*

lemma *cdcl_W-new-marked-at-beginning-is-decide*:

assumes *cdcl_W-stgy* $S S'$ **and**
lev: cdcl_W-M-level-inv S **and**
trail $S' = M' @ \text{Marked } L i \# M$ **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: cdcl_W-stgy.induct*)

case (*conflict'* S') **note** $st = \text{this}(1)$ **and** $\text{no-dup} = \text{this}(2)$ **and** $S' = \text{this}(3)$ **and** $S = \text{this}(4)$
have *cdcl_W-M-level-inv* S'

using *full1-cdcl_W-cp-consistent-inv no-dup st* **by** *blast*

then have $\text{Marked } L i \in \text{set } (\text{trail } S')$ **and** $\text{Marked } L i \notin \text{set } (\text{trail } S)$

using *no-dup unfolding S S' cdcl_W-M-level-inv-def* **by** (*auto simp add: rev-image-eqI*)

then have *False*

using *st rtranclp-cdcl_W-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** $o = \text{this}(1)$ **and** $ns = \text{this}(2)$ **and** $st = \text{this}(3)$ **and** $\text{no-dup} = \text{this}(4)$ **and**
 $S' = \text{this}(5)$ **and** $S = \text{this}(6)$

have *cdcl_W-M-level-inv* U

by (*metis (full-types) lev cdcl_W.simps cdcl_W-consistent-inv full-def o other'.hyp(3) rtranclp-cdcl_W-cp-consistent-inv*)

then have $\text{Marked } L i \in \text{set } (\text{trail } U)$ **and** $\text{Marked } L i \notin \text{set } (\text{trail } S)$

using *no-dup unfolding* $S\ S'\ \text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** (*auto simp add: rev-image-eqI*)
 then have *Marked* $L\ i \in \text{set}\ (\text{trail}\ T)$
 using *st rtrancp-cdcl_W-cp-no-more-Marked-lit* **unfolding** *full-def* **by** *blast*
 then show *?case*
 using *cdcl_W-o-no-more-Marked-lit*[*OF o*] $\langle \text{Marked}\ L\ i \notin \text{set}\ (\text{trail}\ S) \rangle\ \text{ns lev}$ **by** *meson*
 qed

lemma *cdcl_W-o-is-decide*:

assumes *cdcl_W-o* $S'\ T$ **and** *cdcl_W-M-level-inv* S'
 $\text{trail}\ T = \text{drop}\ (\text{length}\ M_0)\ M' @ \text{Marked}\ L\ i \# H @ M$ **and**
 $\neg (\exists M'. \text{trail}\ S' = M' @ \text{Marked}\ L\ i \# H @ M)$
 shows *decide* $S'\ T$
 using *assms*
proof (*induction rule:cdcl_W-o-induct-lev2*)
 case (*backtrack* $K\ i\ M1\ M2\ L\ D$)
 then obtain c **where** $\text{trail}\ S' = c @ M2 @ \text{Marked}\ K\ (\text{Suc}\ i) \# M1$
by *auto*
 then show *?case*
 using *backtrack* **by** (*cases* $\text{drop}\ (\text{length}\ M_0)\ M'$) (*auto simp: cdcl_W-M-level-inv-def*)
 next
 case *decide*
 show *?case* **using** *decide-rule*[*of S'*] *decide(1-4)* **by** *auto*
 qed *auto*

lemma *rtrancp-cdcl_W-new-marked-at-beginning-is-decide*:

assumes *cdcl_W-stgy*** $R\ U$ **and**
 $\text{trail}\ U = M' @ \text{Marked}\ L\ i \# H @ M$ **and**
 $\text{trail}\ R = M$ **and**
cdcl_W-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: rtrancp-induct*)
 case *base*
 then show *?case* **by** *auto*
 next
 case (*step* $T\ U$) **note** $st = \text{this}(1)$ **and** $IH = \text{this}(3)$ **and** $s = \text{this}(2)$ **and**
 $U = \text{this}(4)$ **and** $S = \text{this}(5)$ **and** $\text{lev} = \text{this}(6)$
 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** $cp = \text{this}(1)$ **and** $nd = \text{this}(2)$ **and** $W = \text{this}(3)$
 then obtain M_0 **where** $\text{trail}\ W = M_0 @ \text{trail}\ T$ **and** $\text{nmarked: } \forall l \in \text{set}\ M_0. \neg \text{is-marked}\ l$
 using *rtrancp-cdcl_W-cp-dropWhile-trail* **unfolding** *full1-def rtrancp-unfold* **by** *meson*
 then have $MV: M' @ \text{Marked}\ L\ i \# H @ M = M_0 @ \text{trail}\ T$ **unfolding** W **by** *simp*
 then have $V: \text{trail}\ T = \text{drop}\ (\text{length}\ M_0)\ (M' @ \text{Marked}\ L\ i \# H @ M)$
by *auto*
 have $\text{takeWhile}\ (\text{Not}\ o\ \text{is-marked})\ M' = M_0 @ \text{takeWhile}\ (\text{Not}\ o\ \text{is-marked})\ (\text{trail}\ T)$
 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** $\text{length}\ M_0 \leq \text{length}\ M'$

```

    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' @ Marked L i # H @ M = M0 @ 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 rtrancpl-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-rtrancpl rtrancpl.rtrancpl-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:  $cdcl_W\text{-stgy}^{**} S'' T$  and
4:  $no\text{-step } cdcl_W\text{-cp } S'$  and
6:  $trail S'' = \text{Marked } L i \# H @ M$  and
7:  $trail S' = H @ M$  and
8:  $cdcl_W\text{-stgy}^{**} S' T$  and
9:  $cdcl_W\text{-stgy } S' S'''$  and
10:  $cdcl_W\text{-stgy}^{**} S''' T$ 
    by blast
have  $cdcl_W\text{-stgy}^{**} S'' U$  using  $s \langle cdcl_W\text{-stgy}^{**} S'' T \rangle$  by auto
moreover have  $cdcl_W\text{-stgy}^{**} S' U$  using 8 s by auto
moreover have  $cdcl_W\text{-stgy}^{**} S''' U$  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-cdclW-new-marked-at-beginning-is-decide':
  assumes  $cdcl_W\text{-stgy}^{**} R U$  and
  trail  $U = M' @ \text{Marked } L i \# H @ M$  and
  trail  $R = M$  and
   $cdcl_W\text{-M-level-inv } R$ 
  shows  $\exists y y'. cdcl_W\text{-stgy}^{**} R y \wedge cdcl_W\text{-stgy } y y' \wedge \neg (\exists c. trail y = c @ \text{Marked } L i \# H @ M)$ 
     $\wedge (\lambda a b. cdcl_W\text{-stgy } a b \wedge (\exists c. trail a = c @ \text{Marked } L i \# H @ M))^{**} y' U$ 
proof –
  fix  $T'$ 
  obtain  $S' T T'$  where
    st:  $cdcl_W\text{-stgy}^{**} R S'$  and
    decide  $S' T$  and
    TU:  $cdcl_W\text{-stgy}^{**} T U$  and
    no-step  $cdcl_W\text{-cp } S'$  and
    trT:  $trail T = \text{Marked } L i \# H @ M$  and
    trS':  $trail S' = H @ M$  and
    S'U:  $cdcl_W\text{-stgy}^{**} S' U$  and
    S'T':  $cdcl_W\text{-stgy } S' T'$  and
    T'U:  $cdcl_W\text{-stgy}^{**} T' U$ 
    using rtrancp-cdclW-new-marked-at-beginning-is-decide[OF assms] by blast
  have  $n: \neg (\exists c. trail S' = c @ \text{Marked } L i \# H @ M)$  using trS' by auto
  show ?thesis
    using rtrancp-trans[OF st] rtrancp-exists-last-with-prop[of  $cdcl_W\text{-stgy } S' T' -$ 
       $\lambda a -. \neg (\exists c. trail a = c @ \text{Marked } L i \# H @ M), OF S'T' T'U n]$ 
    by meson
qed

```

```

lemma beginning-not-marked-invert:
  assumes  $A: M @ A = M' @ \text{Marked } K i \# H$  and
  nm:  $\forall m \in set M. \neg is\text{-marked } m$ 
  shows  $\exists M. A = M @ \text{Marked } K i \# H$ 
proof –
  have  $A = drop (length M) (M' @ \text{Marked } K i \# H)$ 
    using arg-cong[OF  $A$ , of  $drop (length M)$ ] by auto
  moreover have  $drop (length M) (M' @ \text{Marked } K i \# H) = drop (length M) M' @ \text{Marked } K i \# H$ 
    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_W-stgy-trail-has-new-marked-is-decide-step*:

assumes *cdcl_W-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_W-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_W-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-cdcl_W-cp-dropWhile-trail' tranclp-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 *cdcl_W-cp*** *T U'* **using** *cp unfolding full-def* **by** *blast*

from *rtranclp-cdcl_W-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*: *cdcl_W-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: cdcl_W-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: cdcl_W-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 *rtrancp-cdcl_W-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: rtrancp-induct*) *auto*

lemma *cdcl_W-o-cannot-learn:*

assumes
cdcl_W-o *y z* **and**
lev: cdcl_W-M-level-inv *y* **and**
trM: trail *y = c @ Marked Kh i # H* **and**
DL: D + {#L#} ∉ # learned-clss *y* **and**
DH: atm-of D ⊆ atm-of 'lits-of H **and**
LH: atm-of L ∉ atm-of 'lits-of H **and**
learned: ∀ T. conflicting y = C-Clause T ⟶ trail y ⊨_{as} CNot T **and**
z: trail z = c' @ Marked Kh i # H

shows *D + {#L#} ∉ # learned-clss z*
using *assms(1-2) trM DL DH LH learned z*

proof (*induction rule: cdcl_W-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_W-M-level-inv-def* **by** *auto*

have *c' @ Marked Kh i # H = Propagated L' (D' + {#L'#}) # trail (reduce-trail-to M1 y)*

using *backtrack.prem(6) decomp undef T lev* **by** (*force simp: cdcl_W-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 ∈ 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'#} ∈ # learned-clss T*

then have *DLD': D + {#L'#} = D' + {#L'#}*

using *DL T neq0-conv undef decomp lev* **by** (*fastforce simp: cdcl_W-M-level-inv-def*)

have *L-cKh: atm-of L ∈ 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_W-M-level-inv-def M3* **by** *auto*

from *arg-cong[OF this, of λa. (Suc j) ∈ set a]* **have** *backtrack-lvl y ≥ j* **by** *auto*

have *DD'[simp]: D = D'*

proof (*rule ccontr*)

assume *D ≠ 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) ≤ 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.. $1 + \text{backtrack-lvl } y$ ]
        using lev unfolding cdclW-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.. $\text{backtrack-lvl } y + 1$ ]
      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.. $\text{backtrack-lvl } y + 1$ ])
        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_W-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-cdcl_W-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-cdcl_W-cp-dropWhile-trail[of T U] **unfolding** *trU full-def* **by** *fastforce*

show ?case
 using cdcl_W-o-cannot-learn[OF o lev trY notin DH LH confl c']
 rtrancpl-cdcl_W-cp-learned-clause-inv cp **unfolding** full-def **by** auto
qed

lemma rtrancpl-cdcl_W-stgy-with-trail-end-has-not-been-learned:
 assumes (λa b. cdcl_W-stgy a b ∧ (∃ c. trail a = c @ Marked K i # H @ []))** S z **and**
 cdcl_W-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: rtrancpl-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 cdcl_W** S T
proof –
 have ∀ p pa. ∃ s sa. ∀ sb sc sd se. (¬ p** (sb::'st) sc ∨ p s sa ∨ pa** sb sc)
 ∧ (¬ pa s sa ∨ ¬ p** sd se ∨ pa** sd se)
by (metis (no-types) mono-rtrancpl)
 then have cdcl_W-stgy** S T
 using st **by** blast
 then show ?thesis
 using rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W **by** blast
qed
 then have lev': cdcl_W-all-struct-inv T
 using rtrancpl-cdcl_W-all-struct-inv-inv[of S T] lev **by** auto
 then have confl': ∀ Ta. conflicting T = C-Clause Ta ⟶ trail T ⊨_{as} CNot Ta
 unfolding cdcl_W-all-struct-inv-def cdcl_W-conflicting-def **by** blast
show ?case
 apply (rule cdcl_W-stgy-with-trail-end-has-not-been-learned[OF - - c - DH LH confl' c'])
 using s lev' IH c **unfolding** cdcl_W-all-struct-inv-def **by** blast+
qed

lemma cdcl_W-stgy-new-learned-clause:
 assumes cdcl_W-stgy S T **and**
 lev: cdcl_W-M-level-inv S **and**
 E ∉# learned-clss S **and**
 E ∈# learned-clss T
 shows ∃ S'. backtrack S S' ∧ conflicting S = C-Clause E ∧ full cdcl_W-cp S' T
 using assms
proof induction
 case conflict'
 then show ?case **unfolding** full1-def **by** (auto dest: trancpl-cdcl_W-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 ∈# 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 ∈# clauses S and
    R: trail R = []
  shows False
proof -
  have M-lev: cdclW-M-level-inv R
    using invR unfolding cdclW-all-struct-inv-def by auto
  have cdclW-M-level-inv S
    using M-lev assms(2) rtrancp-cdclW-stgy-consistent-inv by blast
  with bt obtain D L M1 M2-loc K i where
    T: T ~ 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-True S)))
    and
    decomp: (Marked K (Suc (get-maximum-level D (trail S))) # M1, M2-loc) ∈
      set (get-all-marked-decomposition (trail S)) and
    k: get-level L (trail S) = backtrack-lvl S and
    level: get-level L (trail S) = get-maximum-level (D + {#L#}) (trail S) and
    confl-S: conflicting S = C-Clause (D + {#L#}) and
    i: i = get-maximum-level D (trail S) and
    undef: undefined-lit M1 L
    by (induction rule: backtrack-induction-lev2) metis
  obtain M2 where
    M: trail S = M2 @ Marked K (Suc i) # M1
    using get-all-marked-decomposition-exists-prepend[OF decomp] unfolding i by (metis append-assoc)

  have invS: cdclW-all-struct-inv S
    using invR rtrancp-cdclW-all-struct-inv-inv rtrancp-cdclW-stgy-rtrancp-cdclW st' by blast
  then have confl: cdclW-conflicting S unfolding cdclW-all-struct-inv-def by blast
  then have trail S ⊨as CNot (D + {#L#}) unfolding cdclW-conflicting-def confl-S by auto
  then have MD: trail S ⊨as CNot D by auto

  have lev': cdclW-M-level-inv S using invS unfolding cdclW-all-struct-inv-def by blast

  have get-lvls-M: get-all-levels-of-marked (trail S) = rev [1..W-M-level-inv-def by auto

  have lev: cdclW-M-level-inv R using invR unfolding cdclW-all-struct-inv-def by blast
  then have vars-of-D: atms-of D ⊆ atm-of ' lits-of M1
    using backtrack-atms-of-D-in-M1[OF lev' undef - decomp - - T] confl-S conf T decomp k level
    lev' i undef unfolding cdclW-conflicting-def by (auto simp: cdclW-M-level-inv-def)
  have no-dup (trail S) using lev' by (auto simp: cdclW-M-level-inv-decomp)
  have vars-in-M1:
    ∀ x ∈ atms-of D. x ∉ atm-of ' lits-of (M2 @ [Marked K (get-maximum-level D (trail S) + 1)])

```

apply (*rule vars-of-D distinct-atms-of-incl-not-in-other*[of
 $M2 @ \text{Marked } K \text{ (get-maximum-level } D \text{ (trail } S) + 1) \# [] M1 D]$)
using $\langle \text{no-dup (trail } S) \rangle M \text{ vars-of-D by simp-all}$
have $M1-D: M1 \models_{as} CNot D$
using *vars-in-M1 true-annots-remove-if-notin-vars*[of $M2 @ \text{Marked } K (i + 1) \# [] M1 CNot D$
 $\langle \text{trail } S \models_{as} CNot D \rangle M$ **by** *simp*

have *get-lvls-M*: *get-all-levels-of-marked* (trail S) = *rev* [$1..<Suc \text{ (backtrack-lvl } S)$]
using *lev' unfolding cdcl_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)*)

obtain $M1' K' Ls$ **where**
 M' : trail $S = Ls @ \text{Marked } K' \text{ (backtrack-lvl } S) \# M1'$ **and**
 $Ls: \forall l \in \text{set } Ls. \neg \text{is-marked } l$ **and**
 $\text{set } M1 \subseteq \text{set } M1'$
proof –
let $?Ls = \text{takeWhile (Not o is-marked) (trail } S)$
have $MLs: \text{trail } S = ?Ls @ \text{dropWhile (Not o is-marked) (trail } S)$
by *auto*
have $\text{dropWhile (Not o is-marked) (trail } S) \neq []$ **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**
 $K'k: \text{dropWhile (Not o is-marked) (trail } S)$
 $= \text{Marked } K' K'k \# \text{tl (dropWhile (Not o is-marked) (trail } S))$
by (*cases dropWhile (Not o is-marked) (trail } S);*
cases hd (dropWhile (Not o is-marked) (trail } S)))
simp-all
moreover have $\forall l \in \text{set } ?Ls. \neg \text{is-marked } l$ **using** *set-takeWhileD* **by** *force*
moreover
have *get-all-levels-of-marked* (trail S)
 $= K'k \# \text{get-all-levels-of-marked}(\text{tl (dropWhile (Not o is-marked) (trail } S)))$
apply (*subst MLs, subst K'k*)
using *calculation(2)* **by** (*auto simp add: get-all-levels-of-marked-no-marked*)
then have $K'k = \text{backtrack-lvl } S$
using *calculation(2)* **by** (*auto split: split-if-asm simp add: get-lvls-M upt.simps(2)*)
moreover have $\text{set } M1 \subseteq \text{set (tl (dropWhile (Not o is-marked) (trail } S))}$
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* [$1..<Suc \text{ (backtrack-lvl } S)$]
using *lev' unfolding cdcl_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 $M1'-D: M1' \models_{as} CNot D$ **using** $M1-D \langle \text{set } M1 \subseteq \text{set } M1' \rangle$ **by** (*auto intro: true-annots-mono*)
have $-L \in \text{lits-of (trail } S)$ **using** *conf confl-S* **unfolding** *cdcl_W-conflicting-def* **by** *auto*
have $\text{lvls-}M1': \text{get-all-levels-of-marked } M1' = \text{rev } [1..<\text{backtrack-lvl } S]$
using *get-lvls-M Ls* **by** (*auto simp add: get-all-levels-of-marked-no-marked M'*
split: split-if-asm simp add: upt.simps(2))
have $L \text{-notin: atm-of } L \in \text{atm-of 'lits-of } Ls \vee \text{atm-of } L = \text{atm-of } K'$
proof (*rule ccontr*)
assume $\neg ?thesis$

then have $\text{atm-of } L \notin \text{atm-of ' lits-of (Marked } K' (\text{backtrack-lvl } S) \# \text{ rev } Ls) \text{ by simp}$
 then have $\text{get-level } L (\text{trail } S) = \text{get-level } L M1' \text{ by}$
 unfolding M' by *auto*
 then show *False* using $\text{get-level-in-levels-of-marked}[\text{of } L M1'] \langle \text{backtrack-lvl } S > 0 \rangle$
 unfolding $k \text{ lvs-}M1'$ by *auto*
 qed
 obtain $Y Z$ where
 $RY: \text{cdcl}_W\text{-stgy}^{**} R Y$ and
 $YZ: \text{cdcl}_W\text{-stgy } Y Z$ and
 $nt: \neg (\exists c. \text{trail } Y = c @ \text{Marked } K' (\text{backtrack-lvl } S) \# M1' @ [])$ and
 $Z: (\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Marked } K' (\text{backtrack-lvl } S) \# M1' @ []))^{**}$
 $Z S$
 using $\text{rtrancpl-cdcl}_W\text{-new-marked-at-beginning-is-decide}'[OF \text{ st' - - lev, of } Ls K'$
 $\text{backtrack-lvl } S M1' []]$
 unfolding $R M'$ by *auto*
 have $[\text{simp}]: \text{cdcl}_W\text{-M-level-inv } Y$
 using $RY \text{ lev rtrancpl-cdcl}_W\text{-stgy-consistent-inv}$ by *blast*
 obtain M' where $\text{tr}Z: \text{trail } Z = M' @ \text{Marked } K' (\text{backtrack-lvl } S) \# M1'$
 using $\text{rtrancpl-cdcl}_W\text{-stgy-with-trail-end-has-trail-end}[OF Z] M'$ by *auto*
 have $\text{no-dup } (\text{trail } Y)$
 using $RY \text{ lev rtrancpl-cdcl}_W\text{-stgy-consistent-inv}$ unfolding $\text{cdcl}_W\text{-M-level-inv-def}$ by *blast*
 then obtain Y' where
 $\text{dec: decide } Y Y'$ and
 $Y'Z: \text{full cdcl}_W\text{-cp } Y' Z$ and
 $\text{no-step cdcl}_W\text{-cp } Y$
 using $\text{cdcl}_W\text{-stgy-trail-has-new-marked-is-decide-step}[OF YZ nt Z] M'$ by *auto*
 have $\text{tr}Y: \text{trail } Y = M1'$
 proof –
 obtain M' where $M: \text{trail } Z = M' @ \text{Marked } K' (\text{backtrack-lvl } S) \# M1'$
 using $\text{rtrancpl-cdcl}_W\text{-stgy-with-trail-end-has-trail-end}[OF Z] M'$ by *auto*
 obtain M'' where $M'': \text{trail } Z = M'' @ \text{trail } Y'$ and $\forall m \in \text{set } M''. \neg \text{is-marked } m$
 using $Y'Z \text{ rtrancpl-cdcl}_W\text{-cp-dropWhile-trail'}$ unfolding full-def by *blast*
 obtain M''' where $\text{trail } Y' = M''' @ \text{Marked } K' (\text{backtrack-lvl } S) \# M1'$
 using M'' unfolding M
 by (*metis* (*no-types, lifting*) $\langle \forall m \in \text{set } M''. \neg \text{is-marked } m \rangle \text{ beginning-not-marked-invert}$)
 then show *?thesis* using dec nt by (*induction* M''') *auto*
 qed
 have $Y\text{-CT}: \text{conflicting } Y = C\text{-True}$ using $\langle \text{decide } Y Y' \rangle$ by *auto*
 have $\text{cdcl}_W^{**} R Y$ by (*simp add: RY rtrancpl-cdcl}_W\text{-stgy-rtrancpl-cdcl}_W*)
 then have $\text{init-clss } Y = \text{init-clss } R$ using $\text{rtrancpl-cdcl}_W\text{-init-clss}[\text{of } R Y] M\text{-lev}$ by *auto*
 { assume $DL: D + \{\#L\# \} \in \# \text{ clauses } Y$
 have $\text{atm-of } L \notin \text{atm-of ' lits-of } M1$
 apply (*rule backtrack-lit-skipped*[*of - S*])
 using *decomp i k lev'* unfolding $\text{cdcl}_W\text{-M-level-inv-def}$ by *auto*
 then have $LM1: \text{undefined-lit } M1 L$
 by (*metis* *Marked-Propagated-in-iff-in-lits-of atm-of-uminus image-eqI*)
 have $L\text{-tr}Y: \text{undefined-lit } (\text{trail } Y) L$
 using $L\text{-notin } \langle \text{no-dup } (\text{trail } S) \rangle$ unfolding $\text{defined-lit-map tr}Y M'$
 by (*auto simp add: image-iff lits-of-def*)
 have $\exists Y'. \text{propagate } Y Y'$
 using $\text{propagate-rule}[\text{of } Y] DL M1'\text{-D } L\text{-tr}Y Y\text{-CT tr}Y DL$ by (*metis state-eq-ref*)
 then have *False* using $\langle \text{no-step cdcl}_W\text{-cp } Y \rangle \text{ propagate'}$ by *blast*
 }
 moreover {
 assume $DL: D + \{\#L\# \} \notin \# \text{ clauses } Y$

```

have lY-lZ: learned-clss Y = learned-clss Z
  using dec Y'Z rtranclp-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-rtranclp rtranclp-cdclW-all-struct-inv-inv
      rtranclp-cdclW-stgy-rtranclp-cdclW)
have D + {#L#} ∉ learned-clss S
  apply (rule rtranclp-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) ⟨set M1 ⊆ set M1'⟩ image-mono order-trans
      vars-of-D lits-of-def)
  using L-notin ⟨no-dup (trail S)⟩ 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: ⟨init-clss Y = init-clss R⟩ clauses-def confl-S
      rtranclp-cdclW-stgy-no-more-init-clss)
}
ultimately show False by blast
qed

```

lemma *rtranclp-cdcl_W-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: tranclp-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: rtranclp-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 rtranclp-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⟩
    end
  end

```

```

    by (induction rule: backtrack-induction-lev2) (auto simp: cdclW-M-level-inv-decomp)
  then have  $E \notin \# \text{ clauses } S$ 
    using cdclW-stgy-no-relearned-clause  $R$  inv $R$  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 cdclW-measure where
  cdclW-measure  $S$  =
    [( $\exists :: \text{nat}$ )  $\wedge$  (card (atms-of-msu (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-msu (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  $S$ )  $\leq$  card (atms-of-msu (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 cdclW-termination =
  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::equal  $\Rightarrow$  ('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 :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause conflicting-clause  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st

```

begin

lemma *learned-clss-less-upper-bound:*

fixes $S :: 'st$

assumes

distinct-cdcl_W-state S and

$\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$

shows $\text{card}(\text{set-mset}(\text{learned-clss } S)) \leq 3 \wedge \text{card}(\text{atms-of-msu}(\text{learned-clss } S))$

proof –

have $\text{set-mset}(\text{learned-clss } S) \subseteq \text{build-all-simple-clss}(\text{atms-of-msu}(\text{learned-clss } S))$

apply (*rule simplified-in-build-all*)

using *assms unfolding distinct-cdcl_W-state-def* **by** *auto*

then have $\text{card}(\text{set-mset}(\text{learned-clss } S))$

$\leq \text{card}(\text{build-all-simple-clss}(\text{atms-of-msu}(\text{learned-clss } S)))$

by (*simp add: build-all-simple-clss-finite card-mono*)

then show *?thesis*

by (*meson atms-of-ms-finite build-all-simple-clss-card finite-set-mset order-trans*)

qed

lemma *lexn3[intro!, simp]:*

$a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$

$\implies ([a::\text{nat}, b, c], [a', b', c']) \in \text{lexn } \{(x, y). x < y\} \text{ } 3$

apply *auto*

unfolding *lexn-conv* **apply** *fastforce*

unfolding *lexn-conv* **apply** *auto*

apply (*metis append.simps(1) append.simps(2)*)**+**

done

lemma *cdcl_W-measure-decreasing:*

fixes $S :: 'st$

assumes

cdcl_W 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_W-M-level-inv S and

no-taut: $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$ and

no-dup: distinct-cdcl_W-state S and

confl: cdcl_W-conflicting S

shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$

using *assms(1) M-level assms(2,3)*

proof (*induct rule: cdcl_W-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_W-M-level-inv-decomp(2) marked-lit.sel(2) propagate'*

r-into-rtranclp rtranclp-cdcl_W-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 cdclW.propagate cdclW-no-strange-atm-inv propa M-level by blast
  then have atm-of ' lits-of (Propagated L ( (C + {#L#})) # trail S)
    ⊆ atms-of-msu (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))
    ≤ card (atms-of-msu (init-clss S))
    by (meson atms-of-ms-finite card-mono finite-set-mset)
  then have length (Propagated L ( (C + {#L#})) # trail S) ≤ card (atms-of-msu ?N)
    using no-dup-length-eq-card-atm-of-lits-of no-dup' by fastforce
  then have H: card (atms-of-msu (init-clss S)) - length (trail S)
    = Suc (card (atms-of-msu (init-clss S)) - Suc (length (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-msu (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  $\forall s \in \# \text{learned-clss } ?S'. \neg \text{tautology } s$ 
    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))  $\leq 3 \wedge$  card (atms-of-msu (learned-clss T))
    by (auto simp: clauses-def learned-clss-less-upper-bound)
then have H: card (set-mset ({#D + {#L#}#} + learned-clss S))
     $\leq 3 \wedge$  card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
    using T undef decomp lev by (auto simp: cdclW-M-level-inv-decomp)
moreover
    have atms-of-msu ({#D + {#L#}#} + learned-clss S)  $\subseteq$  atms-of-msu (init-clss S)
    using alien conf unfolding no-strange-atm-def by auto
    then have card-f: card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
         $\leq$  card (atms-of-msu (init-clss S))
    by (meson atms-of-ms-finite card-mono finite-set-mset)
    then have (3::nat)  $\wedge$  card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
         $\leq 3 \wedge$  card (atms-of-msu (init-clss S)) by simp
ultimately have (3::nat)  $\wedge$  card (atms-of-msu (init-clss S))
     $\geq$  card (set-mset ({#D + {#L#}#} + 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)  $\in \text{lexn } \{(a, b). a < b\} \ 3$ 
    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)  $\in \text{lexn } \{(a, b). a < b\} \ 3$ 
    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) ∈ lexn {(a, b). a < b} 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::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) ∈ lexn {(a, b). a < b} 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) ∈ lern {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) ∈ lern {a. case a of (a, b) ⇒ a < b} 3
  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: ∀ T. conflicting S = C-Clause T ⟶ T ∉ # 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 ⟨cdclW-all-struct-inv S⟩ 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-cdcl_W-stgy-decreasing*:
fixes $R\ S\ T :: 'st$
assumes $cdcl_W\text{-stgy}^{++}\ R\ S$
trail $R = []$ **and**
 $cdcl_W\text{-all-struct-inv}\ R$
shows $(cdcl_W\text{-measure}\ S,\ cdcl_W\text{-measure}\ R) \in le_{rn}\ \{(a,\ b).\ a < b\}\ \exists$
using *assms*
apply *induction*
using $cdcl_W\text{-stgy-step-decreasing}[of\ R - R]$ **apply** *blast*
using $cdcl_W\text{-stgy-step-decreasing}[of\ - - R]$ *tranclp-into-rtranclp* $[of\ cdcl_W\text{-stgy}\ R]$
le_{rn}-transI $[OF\ trans\text{-le},\ of\ \exists]$ **unfolding** *trans-def* **by** *blast*

lemma *tranclp-cdcl_W-stgy-S0-decreasing*:
fixes $R\ S\ T :: 'st$
assumes *pl*: $cdcl_W\text{-stgy}^{++}\ (init\text{-state}\ N)\ S$ **and**
no-dup: $distinct\text{-mset-mset}\ N$
shows $(cdcl_W\text{-measure}\ S,\ cdcl_W\text{-measure}\ (init\text{-state}\ N)) \in le_{rn}\ \{(a,\ b).\ a < b\}\ \exists$
proof –
have $cdcl_W\text{-all-struct-inv}\ (init\text{-state}\ N)$
using *no-dup* **unfolding** $cdcl_W\text{-all-struct-inv-def}$ **by** *auto*
then show *?thesis* **using** *pl* *tranclp-cdcl_W-stgy-decreasing* *init-state-trail* **by** *blast*
qed

lemma *wf-tranclp-cdcl_W-stgy*:
 $wf\ \{(S :: 'st,\ init\text{-state}\ N) \mid S\ N.\ distinct\text{-mset-mset}\ N \wedge cdcl_W\text{-stgy}^{++}\ (init\text{-state}\ N)\ S\}$
apply $(rule\ wf\text{-wf-if-measure}'\text{-notation2}[of\ le_{rn}\ \{(a,\ b).\ a < b\}\ \exists - - cdcl_W\text{-measure}])$
apply $(simp\ add:\ wf\ wf\text{-le}_{rn})$
using *tranclp-cdcl_W-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* $[iff]$:
 $(\forall c \in set\ C.\ -c \in lits\text{-of}\ Ms) \longleftrightarrow Ms \models_{as}\ CNot\ (mset\ C)$
unfolding *true-annots-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* $:: 'a\ literal\ list \Rightarrow ('a,\ 'b,\ 'c)\ marked\text{-lit}\ list \Rightarrow 'a\ literal\ option$
where
 $is\text{-unit-clause}\ l\ M =$
 $(case\ List.filter\ (\lambda a.\ atm\text{-of}\ a \notin atm\text{-of}\ 'lits\text{-of}\ M)\ l\ of$
 $\quad a\ \# [] \Rightarrow if\ M \models_{as}\ CNot\ (mset\ l - \{\#a\# \})\ then\ Some\ a\ else\ None$
 $\quad | - \Rightarrow None)$

definition *is-unit-clause-code* :: 'a literal list \Rightarrow ('a, 'b, 'c) marked-lit list
 \Rightarrow 'a literal option **where**
is-unit-clause-code l M =
 (case List.filter (λa . atm-of a \notin atm-of ' lits-of M) l of
 a # [] \Rightarrow if ($\forall c \in \text{set } (\text{remove1 } a \text{ l}). -c \in \text{lits-of } M$) then Some a else None
 | - \Rightarrow None)

lemma *is-unit-clause-is-unit-clause-code*[code]:

is-unit-clause l M = *is-unit-clause-code* l M

proof –

have 1: $\bigwedge a. (\forall c \in \text{set } (\text{remove1 } a \text{ l}). -c \in \text{lits-of } M) \longleftrightarrow M \models_{\text{as}} \text{CNot } (\text{mset } l - \{\#a\# \})$

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 = Some a

shows *undefined-lit* M a

proof –

have (case [a \leftarrow l . atm-of a \notin atm-of ' lits-of M] of [] \Rightarrow None

| [a] \Rightarrow if M \models_{as} CNot (mset l - {#a#}) then Some a else None

| a # ab # xa \Rightarrow Map.empty xa) = Some a

using *assms* **unfolding** *is-unit-clause-def* .

hence a \in set [a \leftarrow l . atm-of a \notin atm-of ' lits-of M]

apply (case-tac [a \leftarrow l . atm-of a \notin atm-of ' lits-of M])

apply *simp*

apply (case-tac list) **by** (auto split: split-if-asm)

hence atm-of a \notin atm-of ' lits-of M **by** auto

thus ?thesis

by (*simp* add: Marked-Propagated-in-iff-in-lits-of
 atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)

qed

lemma *is-unit-clause-some-CNot*: *is-unit-clause* l M = Some a \implies M \models_{as} CNot (mset l - {#a#})

unfolding *is-unit-clause-def*

proof –

assume (case [a \leftarrow l . atm-of a \notin atm-of ' lits-of M] of [] \Rightarrow None

| [a] \Rightarrow if M \models_{as} CNot (mset l - {#a#}) then Some a else None

| a # ab # xa \Rightarrow Map.empty xa) = Some a

thus ?thesis

apply (case-tac [a \leftarrow l . atm-of a \notin 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*: *is-unit-clause* l M = Some a \implies a \in set l

unfolding *is-unit-clause-def*

proof –

assume (case [a \leftarrow l . atm-of a \notin atm-of ' lits-of M] of [] \Rightarrow None

| [a] \Rightarrow if M \models_{as} CNot (mset l - {#a#}) then Some a else None

| a # ab # xa \Rightarrow Map.empty xa) = Some a

thus a \in set l

by (case-tac [a \leftarrow l . atm-of a \notin atm-of ' lits-of M])

(fastforce dest: filter-eq-ConsD split: split-if-asm split: list.splits)+
qed

lemma *is-unit-clause-nil[simp]*: *is-unit-clause* [] *M* = *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
 None \Rightarrow *find-first-unit-clause* *l* *M*
 | Some *L* \Rightarrow Some (*L*, *a*)) |
find-first-unit-clause [] - = *None*

lemma *find-first-unit-clause-some*:
find-first-unit-clause *l* *M* = Some (*a*, *c*)
 $\implies c \in \text{set } l \wedge M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\}) \wedge \text{undefined-lit } M a \wedge a \in \text{set } c$
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 *None*

proof -

have [*a* $\leftarrow c$. *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*: *mset* *c* + {*#ac*#} - {*#a*#} = *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*:

distinct c $\implies c \in \text{set } l \implies M \models_{as} CNot (mset\ c - \{\#a\# \}) \implies \text{undefined-lit } M\ a \implies a \in \text{set } c$
 $\implies \text{find-first-unit-clause } l\ M \neq None$

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 ($\lambda lit. lit \notin M \wedge \neg lit \notin M$) a of*

None $\Rightarrow \text{find-first-unused-var } l\ M$

| Some a $\Rightarrow \text{Some a}$ |

find-first-unused-var [] - = None

lemma *find-none[iff]*:

List.find ($\lambda lit. lit \notin M \wedge \neg lit \notin M$) a = None $\longleftrightarrow \text{atm-of 'set } a \subseteq \text{atm-of ' } M$

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 ($\lambda lit. lit \notin M \wedge \neg lit \notin M$) a = Some b $\implies b \in \text{set } a \wedge b \notin M \wedge \neg b \notin M$*

unfolding *find-Some-iff* **by** (*metis nth-mem*)

lemma *find-first-unused-var-None[iff]*:

find-first-unused-var l M = None $\longleftrightarrow (\forall a \in \text{set } l. \text{atm-of 'set } a \subseteq \text{atm-of ' } M)$

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 $\neg(\forall a \in \text{set } l. \text{atm-of 'set } a \subseteq \text{atm-of ' } M)$

proof –

have *find-first-unused-var l M $\neq None$*

using *assms* **by** (*cases find-first-unused-var l M*) *auto*

thus $\neg(\forall a \in \text{set } l. \text{atm-of 'set } a \subseteq \text{atm-of ' } M)$ **by** *auto*

qed

lemma *find-first-unused-var-Some*:

find-first-unused-var l M = Some a $\implies (\exists m \in \text{set } l. a \in \text{set } m \wedge a \notin M \wedge \neg a \notin M)$

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 $\implies \text{undefined-lit } Ms\ a$

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 $\sim\sim$ /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 $DPLL\text{-}step :: int\ dpll_W\text{-}marked\text{-}lits \times int\ literal\ list\ list$

$\Rightarrow int\ dpll_W\text{-}marked\text{-}lits \times int\ literal\ list\ list$ **where**

$DPLL\text{-}step = (\lambda(Ms, N).$

(*case find-first-unit-clause* $N\ Ms$ of
Some $(L, -) \Rightarrow (Propagated\ L\ () \# Ms, N)$
 | $- \Rightarrow$
if $\exists C \in set\ N. (\forall c \in set\ C. -c \in lits\text{-}of\ Ms)$
then
 (*case backtrack-split* Ms of
 ($-, L \# M) \Rightarrow (Propagated\ (-\ (lits\text{-}of\ L))\ () \# M, N)$
 | $(-, -) \Rightarrow (Ms, N)$
)
else
 (*case find-first-unused-var* $N\ (lits\text{-}of\ Ms)$ of
Some $a \Rightarrow (Marked\ a\ () \# Ms, N)$
 | *None* $\Rightarrow (Ms, N))))$

Example of propagation:

value $DPLL\text{-}step\ ([Marked\ (Neg\ 1)\ ()], [[Pos\ (1::int), Neg\ 2]])$

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation $toS \equiv \lambda(Ms::int, unit, unit)\ marked\text{-}lit\ list$

$(N::int\ literal\ list\ list). (Ms, mset\ (map\ mset\ N))$

abbreviation $toS' \equiv \lambda(Ms::int, unit, unit)\ marked\text{-}lit\ list,$

$N::int\ literal\ list\ list). (Ms, mset\ (map\ mset\ N))$

Proof of correctness of $DPLL\text{-}step$

lemma $DPLL\text{-}step\text{-}is\text{-}a\text{-}dpll_W\text{-}step:$

assumes $step: (Ms', N') = DPLL\text{-}step\ (Ms, N)$

and $neg: (Ms, N) \neq (Ms', N')$

shows $dpll_W\ (toS\ Ms\ N)\ (toS\ Ms'\ N')$

proof –

let $?S = (Ms, mset\ (map\ mset\ N))$

{ fix $L\ E$

assume $unit: find\text{-}first\text{-}unit\text{-}clause\ N\ Ms = Some\ (L, E)$

hence $Ms'N: (Ms', N') = (Propagated\ L\ () \# Ms, N)$

using $step$ **unfolding** $DPLL\text{-}step\text{-}def$ **by** $auto$

obtain C **where**

$C: C \in set\ N$ **and**

$Ms: Ms \models_{as} CNot\ (mset\ C - \{\#L\#})$ **and**

$undef: undefined\text{-}lit\ Ms\ L$ **and**

$L \in set\ C$ **using** $find\text{-}first\text{-}unit\text{-}clause\text{-}some[OF\ unit]$ **by** $metis$

have $dpll_W\ (Ms, mset\ (map\ mset\ N))$

$(Propagated\ L\ () \# fst\ (Ms, mset\ (map\ mset\ N)), snd\ (Ms, mset\ (map\ mset\ N)))$

apply $(rule\ dpll_W.propagate)$

using $Ms\ undef\ C\ (L \in set\ C)$ **unfolding** $mem\text{-}set\text{-}multiset\text{-}eq$ **by** $(auto\ simp\ add: C)$

hence $?thesis$ **using** $Ms'N$ **by** $auto$

}

moreover

{ assume $unit: find\text{-}first\text{-}unit\text{-}clause\ N\ Ms = None$

assume $exC: \exists C \in set\ N. Ms \models_{as} CNot\ (mset\ C)$

then obtain C where $C: C \in \text{set } N$ and $Ms: Ms \models_{as} C \text{Not } (mset\ C)$ by *auto*
 then obtain $L\ M\ M'$ where $bt: \text{backtrack-split } Ms = (M', L \# M)$
 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: $dpll_W\ (Ms, mset\ (map\ mset\ N))$
 $(\text{Propagated } (-\ \text{lit-of } L)\ () \# M, \text{snd } (Ms, mset\ (map\ mset\ N)))$
 apply *(rule dpll_W.backtrack[OF - (is-marked L), of])*
 using $C\ Ms\ bt$ by *auto*
 moreover have $(Ms', N') = (\text{Propagated } (-\ (\text{lit-of } L))\ () \# M, N)$
 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: $\neg (\exists C \in \text{set } N. Ms \models_{as} C \text{Not } (mset\ C))$*
 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 $dpll_W\ (Ms, mset\ (map\ mset\ N))$
 $(\text{Marked } L\ () \# \text{fst } (Ms, mset\ (map\ mset\ N)), \text{snd } (Ms, mset\ (map\ mset\ N)))$
 apply *(rule dpll_W.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-ms-def)*
 moreover have $(Ms', N') = (\text{Marked } L\ () \# Ms, N)$
 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: $(Ms, N) = \text{DPLL-step } (Ms, N)$*
 shows *conclusive-dpll_W-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: $\exists C \in \text{set } N. Ms \models_{as} C \text{Not } (mset\ C)$*
 hence $Ms: (Ms, N) = (\text{case } \text{backtrack-split } Ms \text{ of } (x, []) \Rightarrow (Ms, N) \mid (x, L \# M) \Rightarrow (\text{Propagated } (-\ \text{lit-of } L)\ () \# M, N))$
 using *step unfolding DPLL-step-def* by *(simp add:unit)*

have $\text{snd } (\text{backtrack-split } Ms) = []$

proof *(cases backtrack-split Ms, cases snd (backtrack-split Ms))*

fix $a\ b$

assume *backtrack-split Ms = (a, b)* and $\text{snd } (\text{backtrack-split } Ms) = []$

thus $\text{snd } (\text{backtrack-split } Ms) = []$ by *blast*

next

fix $a\ b\ aa\ list$

assume

bt: backtrack-split Ms = (a, b) and

bt': $\text{snd } (\text{backtrack-split } Ms) = aa \# list$

hence $Ms: Ms = \text{Propagated } (-\ \text{lit-of } aa)\ () \# list$ 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: ¬ (∃ C ∈ set N. Ms ⊨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: ∀ a ∈ set N. atm-of 'set a ⊆ atm-of ' (lits-of Ms) by auto
  have fst (toS Ms N) ⊨asm snd (toS Ms N) unfolding true-annots-def CNot-def Ball-def
  proof clarify
    fix x
    assume x: x ∈ set-mset (clauses (toS Ms N))
    hence ¬Ms ⊨as CNot x using n unfolding true-annots-def CNot-def Ball-def by auto
    moreover have total-over-m (lits-of Ms) {x}
      using a x image-iff in-mono atms-of-s-def
      unfolding total-over-m-def total-over-set-def lits-of-def by fastforce
    ultimately show fst (toS Ms N) ⊨ x
      using total-not-CNot[of lits-of Ms x] by (simp add: true-annot-def true-annots-true-cls)
    qed
  hence ?thesis unfolding conclusive-dpllW-state-def by blast
}
ultimately show ?thesis by blast
qed

```

18.2.2 Adding invariants

Invariant tested in the function `function DPLL-ci :: int dpllW-marked-lits ⇒ int literal list list`

```

⇒ int dpllW-marked-lits × int literal list list where
DPLL-ci Ms N =
  (if ¬dpllW-all-inv (Ms, mset (map mset N))
   then (Ms, N)
   else
    let (Ms', N') = DPLL-step (Ms, N) in
    if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-ci Ms' N)
  by fast+

```

termination

```

proof (relation {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}})
  show wf {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}
    using wf-if-measure-f[OF dpllW-wf, of toS'] by auto

```

next

```

fix Ms :: int dpllW-marked-lits and N x xa y
assume ¬ ¬ dpllW-all-inv (toS Ms N)
and step: x = DPLL-step (Ms, N)
and x: (xa, y) = x
and (xa, y) ≠ (Ms, N)
thus ((xa, N), Ms, N) ∈ {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}
  using DPLL-step-is-a-dpllW-step dpllW-same-clauses split-conv by fastforce
qed

```

No invariant tested **function** (*domintros*) *DPLL-part*:: *int dpll_W-marked-lits* \Rightarrow *int literal list list*
 \Rightarrow
int dpll_W-marked-lits \times *int literal list list* **where**
DPLL-part Ms N =
 (*let* (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*) *in*
 if (*Ms'*, *N'*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else* *DPLL-part Ms' N*)
by *fast+*

lemma *snd-DPLL-step[simp]*:
snd (*DPLL-step* (*Ms*, *N*)) = *N*
unfolding *DPLL-step-def* **by** (*auto split: split-if option.splits prod.splits list.splits*)

lemma *dpll_W-all-inv-implieS-2-eq3-and-dom*:
assumes *dpll_W-all-inv* (*Ms*, *mset* (*map mset N*))
shows *DPLL-ci Ms N* = *DPLL-part Ms N* \wedge *DPLL-part-dom* (*Ms*, *N*)
using *assms*

proof (*induct rule: DPLL-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'*: *dpll_W-all-inv* (*toS Ms' N*) **by** (*metis* (*mono-tags*) *1.prem*s *DPLL-step-is-a-dpll_W-step Ms'*
dpll_W-all-inv old.prod.inject)
{ assume (*Ms'*, *N*) \neq (*Ms*, *N*)
 hence *DPLL-ci Ms' N* = *DPLL-part Ms' N* \wedge *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*s *DPLL-part.psimps Ms'*
 (*DPLL-ci Ms' N* = *DPLL-part Ms' N* \wedge *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-dpll_W-rtrancp*:
assumes *DPLL-ci Ms N* = (*Ms'*, *N'*)
shows *dpll_W*** (*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 \neg *dpll_W-all-inv* (*toS Ms N*)
 hence (*Ms*, *N*) = (*Ms'*, *N*) **using** *step* **by** *auto*
 hence *?case* **by** *auto*
}
moreover
{ assume *dpll_W-all-inv* (*toS Ms N*)
 and (*S₁*, *S₂*) = (*Ms*, *N*)
 hence *?case* **using** *S step* **by** *auto*

```

}
moreover
{ assume  $dpll_W\text{-all-inv}$  (toS Ms N)
  and  $(S_1, S_2) \neq (Ms, N)$ 
  moreover obtain  $S_1' S_2'$  where  $DPLL\text{-ci } S_1 N = (S_1', S_2')$  by (case-tac  $DPLL\text{-ci } S_1 N$ ) auto
  moreover have  $DPLL\text{-ci } Ms N = DPLL\text{-ci } S_1 N$  using  $DPLL\text{-ci.simps}$ [of Ms N] calculation
  proof –
    have (case  $(S_1, S_2)$  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 by presburger
    hence (if  $(S_1, S_2) = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-ci } S_1 N = DPLL\text{-ci } Ms N$ 
      by fastforce
    thus ?thesis
    using calculation(2) by presburger
  qed
ultimately have  $dpll_W^{**}$  (toS  $S_1' N$ ) (toS Ms' N) using IH[of  $(S_1, S_2) S_1 S_2$ ] S step by simp

moreover have  $dpll_W$  (toS Ms N) (toS  $S_1 N$ )
  by (metis  $DPLL\text{-step-is-a-dpll}_W\text{-step } S \langle (S_1, S_2) \neq (Ms, N) \rangle$  prod.sel(2) snd-DPLL-step)
ultimately have ?case by (metis (mono-tags, hide-lams) IH S  $\langle (S_1, S_2) \neq (Ms, N) \rangle$ 
   $\langle DPLL\text{-ci } Ms N = DPLL\text{-ci } S_1 N \rangle \langle dpll_W\text{-all-inv (toS Ms N) \rangle$  converse-rtranclp-into-rtranclp
  local.step)
}
ultimately show ?case by blast
qed

lemma  $dpll_W\text{-all-inv-dpll}_W\text{-tranclp-irrefl}$ :
  assumes  $dpll_W\text{-all-inv}$  (Ms, N)
  and  $dpll_W^{++}$  (Ms, N) (Ms, N)
  shows False
proof –
  have 1: wf  $\{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W^{++} S S'\}$  using  $dpll_W\text{-wf-tranclp}$  by auto
  have  $((Ms, N), (Ms, N)) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W^{++} S S'\}$  using assms by auto
  thus False using wf-not-refl[OF 1] by blast
qed

lemma  $DPLL\text{-ci-final-state}$ :
  assumes step:  $DPLL\text{-ci } Ms N = (Ms, N)$ 
  and inv:  $dpll_W\text{-all-inv}$  (toS Ms N)
  shows conclusive- $dpll_W\text{-state}$  (toS Ms N)
proof –
  have st:  $dpll_W^{**}$  (toS Ms N) (toS Ms N) using  $DPLL\text{-ci-dpll}_W\text{-rtranclp}$ [OF step] .
  have  $DPLL\text{-step}$  (Ms, N) = (Ms, N)
  proof (rule ccontr)
    obtain Ms' N' where Ms'N:  $(Ms', N') = DPLL\text{-step}$  (Ms, N)
    by (case-tac  $DPLL\text{-step}$  (Ms, N)) auto
    assume  $\neg$  ?thesis
    hence  $DPLL\text{-ci } Ms' N = (Ms, N)$  using step inv st Ms'N[symmetric] by fastforce
    hence  $dpll_W^{++}$  (toS Ms N) (toS Ms N)
    by (metis  $DPLL\text{-ci-dpll}_W\text{-rtranclp } DPLL\text{-step-is-a-dpll}_W\text{-step } Ms'N \langle DPLL\text{-step (Ms, N) } \neq (Ms, N) \rangle$ 
      prod.sel(2) rtranclp-into-tranclp2 snd-DPLL-step)
    thus False using  $dpll_W\text{-all-inv-dpll}_W\text{-tranclp-irrefl inv}$  by auto
  qed
  thus ?thesis using  $DPLL\text{-step-stuck-final-state}$ [of Ms N] by simp

```

qed

lemma *DPLL-step-obtains*:

obtains Ms' where $(Ms', N) = DPLL\text{-}step\ (Ms, N)$
 unfolding *DPLL-step-def* by (metis (no-types, lifting) *DPLL-step-def prod.collapse snd-DPLL-step*)

lemma *DPLL-ci-obtains*:

obtains Ms' where $(Ms', N) = DPLL\text{-}ci\ Ms\ N$

proof (induct rule: *DPLL-ci.induct*)

case (1 $Ms\ N$) note $IH = this(1)$ and $that = this(2)$

obtain S where $SN: (S, N) = DPLL\text{-}step\ (Ms, N)$ using *DPLL-step-obtains* by metis

{ assume $\neg dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$

hence ?case using *that* by auto

}

moreover {

assume $n: (S, N) \neq (Ms, N)$

and $inv: dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$

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-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-step-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-ci-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-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_W-all-inv-final*:

```

  fixes M Ms': (int, unit, unit) marked-lit list and
    N :: int literal list list
  assumes inv: dpllW-all-inv (Ms, mset (map mset N))
  and MsN: DPLL-part Ms N = (Ms', N)
  shows conclusive-dpllW-state (toS Ms' N) ∧ dpllW** (toS Ms N) (toS Ms' N)
proof -
  have 2: DPLL-ci Ms N = DPLL-part Ms N using inv dpllW-all-inv-implicS-2-eq3-and-dom by blast
  hence star: dpllW** (toS Ms N) (toS Ms' N) unfolding MsN using DPLL-ci-dpllW-rtranclp by blast
  hence inv': dpllW-all-inv (toS Ms' N) using inv rtranclp-dpllW-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 dpllW-state =`

```

  {(M::(int, unit, unit) marked-lit list, N::int literal list list).
   dpllW-all-inv (toS M N)}
```

`morphisms rough-state-of state-of`

proof

```

  show ([], []) ∈ {(M, N). dpllW-all-inv (toS M N)} by (auto simp add: dpllW-all-inv-def)

```

qed

lemma

```

  DPLL-part-dom ([], N)
```

```

  using assms dpllW-all-inv-implicS-2-eq3-and-dom[of [] N] by (simp add: dpllW-all-inv-def)

```

Some type classes `instantiation dpllW-state :: equal`

begin

```

definition equal-dpllW-state :: dpllW-state ⇒ dpllW-state ⇒ bool where
  equal-dpllW-state S S' = (rough-state-of S = rough-state-of S')
```

instance

```

  by standard (simp add: rough-state-of-inject equal-dpllW-state-def)

```

end

DPLL **definition** *DPLL-step'* :: dpll_W-state ⇒ dpll_W-state **where**

```

  DPLL-step' S = state-of (DPLL-step (rough-state-of S))

```

declare *rough-state-of-inverse*[simp]

lemma *DPLL-step-dpll_W-conc-inv*:

$DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S) \in \{(M, N). \text{dpll}_W\text{-all-inv}\ (toS\ M\ N)\}$
by (smt $DPLL\text{-}ci.simps\ DPLL\text{-}ci\text{-}dpll_W\text{-}rtrancplp\ case\text{-}prodE\ case\text{-}prodI2\ rough\text{-}state\text{-}of\ mem\text{-}Collect\text{-}eq\ old.prod.case\ prod.sel(2)\ rtrancplp\text{-}dpll_W\text{-}all\text{-}inv\ snd\text{-}DPLL\text{-}step$)

lemma $rough\text{-}state\text{-}of\text{-}DPLL\text{-}step'\text{-}DPLL\text{-}step[simp]$:
 $rough\text{-}state\text{-}of\ (DPLL\text{-}step'\ S) = DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S)$
using $DPLL\text{-}step\text{-}dpll_W\text{-}conc\text{-}inv\ DPLL\text{-}step'\text{-}def\ state\text{-}of\text{-}inverse$ **by** *auto*

function $DPLL\text{-}tot::\ dpll_W\text{-}state \Rightarrow dpll_W\text{-}state$ **where**
 $DPLL\text{-}tot\ S =$
 $(let\ S' = DPLL\text{-}step'\ S\ in$
 $\quad if\ S' = S\ then\ S\ else\ DPLL\text{-}tot\ S')$
by *fast+*

termination
proof (relation $\{(T', T).$
 $(rough\text{-}state\text{-}of\ T', rough\text{-}state\text{-}of\ T)$
 $\in \{(S', S). (toS'\ S', toS'\ S)$
 $\in \{(S', S). \text{dpll}_W\text{-all-inv}\ S \wedge \text{dpll}_W\ S\ S'\}\})$
show $wf\ \{(b, a).$
 $(rough\text{-}state\text{-}of\ b, rough\text{-}state\text{-}of\ a)$
 $\in \{(b, a). (toS'\ b, toS'\ a)$
 $\in \{(b, a). \text{dpll}_W\text{-all-inv}\ a \wedge \text{dpll}_W\ a\ b\}\})$
using $wf\text{-}if\text{-}measure\text{-}f[OF\ wf\text{-}if\text{-}measure\text{-}f[OF\ \text{dpll}_W\text{-}wf,\ of\ toS'],\ of\ rough\text{-}state\text{-}of]$.
next
fix $S\ x$
assume $x: x = DPLL\text{-}step'\ S$
and $x \neq S$
have $\text{dpll}_W\text{-all-inv}\ (case\ rough\text{-}state\text{-}of\ S\ of\ (Ms, N) \Rightarrow (Ms, mset\ (map\ mset\ N)))$
by (*metis* (*no-types*, *lifting*) $case\text{-}prodE\ mem\text{-}Collect\text{-}eq\ old.prod.case\ rough\text{-}state\text{-}of$)
moreover have $\text{dpll}_W\ (case\ rough\text{-}state\text{-}of\ S\ of\ (Ms, N) \Rightarrow (Ms, mset\ (map\ mset\ N)))$
 $(case\ rough\text{-}state\text{-}of\ (DPLL\text{-}step'\ S)\ of\ (Ms, N) \Rightarrow (Ms, mset\ (map\ mset\ N)))$
proof –
obtain $Ms\ N$ **where** $Ms: (Ms, N) = rough\text{-}state\text{-}of\ S$ **by** (*cases* $rough\text{-}state\text{-}of\ S$) *auto*
have $\text{dpll}_W\text{-all-inv}\ (toS'\ (Ms, N))$ **using** *calculation* **unfolding** Ms **by** *blast*
moreover obtain $Ms'\ N'$ **where** $Ms': (Ms', N') = rough\text{-}state\text{-}of\ (DPLL\text{-}step'\ S)$
by (*cases* $rough\text{-}state\text{-}of\ (DPLL\text{-}step'\ S)$) *auto*
ultimately have $\text{dpll}_W\text{-all-inv}\ (toS'\ (Ms', N'))$ **unfolding** Ms'
by (*metis* (*no-types*, *lifting*) $case\text{-}prod\text{-}unfold\ mem\text{-}Collect\text{-}eq\ rough\text{-}state\text{-}of$)

have $\text{dpll}_W\ (toS\ Ms\ N)\ (toS\ Ms'\ N')$
apply (*rule* $DPLL\text{-}step\text{-}is\text{-}a\text{-}dpll_W\text{-}step[of\ Ms'\ N'\ Ms\ N]$)
unfolding $Ms\ Ms'$ **using** $\langle x \neq S \rangle\ rough\text{-}state\text{-}of\text{-}inject\ x$ **by** *fastforce+*
thus *?thesis* **unfolding** $Ms[symmetric]\ Ms'[symmetric]$ **by** *auto*
qed
ultimately show $(x, S) \in \{(T', T). (rough\text{-}state\text{-}of\ T', rough\text{-}state\text{-}of\ T)$
 $\in \{(S', S). (toS'\ S', toS'\ S) \in \{(S', S). \text{dpll}_W\text{-all-inv}\ S \wedge \text{dpll}_W\ S\ S'\}\})$
by (*auto* *simp* *add: x*)
qed

lemma [*code*]:
 $DPLL\text{-}tot\ S =$
 $(let\ S' = DPLL\text{-}step'\ S\ in$
 $\quad if\ S' = S\ then\ S\ else\ DPLL\text{-}tot\ S')$ **by** *auto*

lemma $DPLL\text{-}tot\text{-}DPLL\text{-}step\text{-}DPLL\text{-}tot[simp]$: $DPLL\text{-}tot\ (DPLL\text{-}step'\ S) = DPLL\text{-}tot\ S$

apply (*cases* $DPLL\text{-}step' S = S$)
apply *simp*
unfolding $DPLL\text{-}tot.simps[of S]$ **by** (*simp del: DPLL-tot.simps*)

lemma $DOPLL\text{-}step'\text{-}DPLL\text{-}tot[simp]$:
 $DPLL\text{-}step' (DPLL\text{-}tot S) = DPLL\text{-}tot S$
by (*rule DPLL-tot.induct[of $\lambda S. DPLL\text{-}step' (DPLL\text{-}tot S) = DPLL\text{-}tot S S]$*)
(metis (full-types) DPLL-tot.simps)

lemma $DPLL\text{-}tot\text{-}final\text{-}state$:
assumes $DPLL\text{-}tot S = S$
shows *conclusive-dpll_W-state* (*toS'* (*rough-state-of* S))
proof –
have $DPLL\text{-}step' S = S$ **using** *assms[symmetric] DOPLL-step'-DPLL-tot* **by** *metis*
hence $DPLL\text{-}step (rough\text{-}state\text{-}of S) = (rough\text{-}state\text{-}of S)$
unfolding $DPLL\text{-}step'\text{-}def$ **using** $DPLL\text{-}step\text{-}dpll_W\text{-}conc\text{-}inv$ *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 $DPLL\text{-}tot\text{-}star$:
assumes $rough\text{-}state\text{-}of (DPLL\text{-}tot S) = S'$
shows $dpll_W^{**} (toS' (rough\text{-}state\text{-}of S)) (toS' S')$
using *assms*
proof (*induction arbitrary: S' rule: DPLL-tot.induct*)
case ($1 S S'$)
let $?x = DPLL\text{-}step' S$
{ assume $?x = S$
then have $?case$ **using** $1(2)$ **by** *simp*
}
moreover {
assume $S: ?x \neq S$
have $?case$
apply (*cases* $DPLL\text{-}step' S = S$)
using S **apply** *blast*
by (*smt 1.IH 1.prem DPLL-step-is-a-dpll_W-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 $rough\text{-}state\text{-}of\text{-}rough\text{-}state\text{-}of\text{-}nil[simp]$:
 $rough\text{-}state\text{-}of (state\text{-}of ([], N)) = ([], N)$
apply (*rule DPLL-W-Implementation.dpll_W-state.state-of-inverse*)
unfolding $dpll_W\text{-}all\text{-}inv\text{-}def$ **by** *auto*

Theorem of correctness

lemma $DPLL\text{-}tot\text{-}correct$:
assumes $rough\text{-}state\text{-}of (DPLL\text{-}tot (state\text{-}of ([], N))) = (M, N')$
and $(M', N'') = toS' (M, N')$
shows $M' \models_{asm} N'' \longleftrightarrow \text{satisfiable } (set\text{-}mset N'')$

proof –

have $dpll_W^{**} (toS' ([], N)) (toS' (M, N'))$ **using** $DPLL-tot-star[OF\ assms(1)]$ **by** *auto*
moreover have $conclusive-dpll_W-state (toS' (M, N'))$
using $DPLL-tot-final-state$ **by** $(metis\ (mono-tags,\ lifting)\ DOPLL-step'-DPLL-tot\ DPLL-tot.simps\ assms(1))$
ultimately show $?thesis$ **using** $dpll_W-conclusive-state-correct$ **by** $(smt\ DPLL-ci.simps\ DPLL-ci-dpll_W-rtrancpl\ assms(2)\ dpll_W-all-inv-def\ prod.case\ prod.sel(1)\ prod.sel(2)\ rtrancpl-dpll_W-inv(3)\ rtrancpl-dpll_W-inv-starting-from-0)$

qed

18.2.3 Code export

A conversion to $DPLL-W$ -Implementation. $dpll_W-state$ **definition** $Con :: (int, unit, unit) \text{ marked-lit } list \times int \text{ literal list list}$

$\Rightarrow dpll_W-state$ **where**

$Con\ xs = state-of\ (if\ dpll_W-all-inv\ (toS\ (fst\ xs)\ (snd\ xs))\ then\ xs\ else\ ([], []))$

lemma $[code\ abstype]:$

$Con\ (rough-state-of\ S) = S$

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** $(metis\ (mono-tags,\ lifting)\ DPLL-step-dpll_W-conc-inv\ mem-Collect-eq\ prod.case-eq-if)$

A slightly different version of $DPLL-tot$ where the returned boolean indicates the result.

definition $DPLL-tot-rep$ **where**

$DPLL-tot-rep\ S =$

$(let\ (M, N) = (rough-state-of\ (DPLL-tot\ S))\ in\ (\forall A \in set\ N.\ (\exists a \in set\ A.\ a \in lits-of\ (M)), M))$

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 $'a\ cdcl_W-mark = 'a\ clause$

type-synonym $cdcl_W-marked-level = nat$

type-synonym $'v\ cdcl_W-marked-lit = ('v,\ cdcl_W-marked-level,\ 'v\ cdcl_W-mark)\ marked-lit$

type-synonym $'v\ cdcl_W-marked-lits = ('v,\ cdcl_W-marked-level,\ 'v\ cdcl_W-mark)\ marked-lits$

type-synonym $'v\ cdcl_W-state =$

$'v \text{ cdcl}_W\text{-marked-lits} \times 'v \text{ clauses} \times 'v \text{ clauses} \times \text{nat} \times 'v \text{ clause conflicting-clause}$

abbreviation $\text{trail} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ where}$
 $\text{trail} \equiv (\lambda(M, -). M)$

abbreviation $\text{cons-trail} :: 'a \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \text{ where}$
 $\text{cons-trail} \equiv (\lambda L (M, S). (L \# M, S))$

abbreviation $\text{tl-trail} :: 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \text{ where}$
 $\text{tl-trail} \equiv (\lambda(M, S). (\text{tl } M, S))$

abbreviation $\text{clauses} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b \text{ where}$
 $\text{clauses} \equiv \lambda(M, N, -). N$

abbreviation $\text{learned-clss} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c \text{ where}$
 $\text{learned-clss} \equiv \lambda(M, N, U, -). U$

abbreviation $\text{backtrack-lvl} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd \text{ where}$
 $\text{backtrack-lvl} \equiv \lambda(M, N, U, k, -). k$

abbreviation $\text{update-backtrack-lvl} :: 'd \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
 $\text{update-backtrack-lvl} \equiv \lambda k (M, N, U, -, S). (M, N, U, k, S)$

abbreviation $\text{conflicting} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e \text{ where}$
 $\text{conflicting} \equiv \lambda(M, N, U, k, D). D$

abbreviation $\text{update-conflicting} :: 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
 $\text{update-conflicting} \equiv \lambda S (M, N, U, k, -). (M, N, U, k, S)$

abbreviation $S0\text{-cdcl}_W \ N \equiv (([], N, \{\#\}, 0, C\text{-True}) :: 'v \text{ cdcl}_W\text{-state})$

abbreviation $\text{add-learned-cls} \text{ where}$
 $\text{add-learned-cls} \equiv \lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

abbreviation $\text{remove-cls} \text{ where}$
 $\text{remove-cls} \equiv \lambda C (M, N, U, S). (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$
interpretation cdcl_W : $\text{state}_W \ \text{trail} \ \text{clauses} \ \text{learned-clss} \ \text{backtrack-lvl} \ \text{conflicting}$
 $\lambda L (M, S). (L \# M, S)$
 $\lambda(M, S). (\text{tl } M, S)$
 $\lambda C (M, N, S). (M, \{\#C\# \} + N, S)$
 $\lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$
 $\lambda C (M, N, U, S). (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$
 $\lambda(k :: \text{nat}) (M, N, U, -, D). (M, N, U, k, D)$
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$
 $\lambda N. ([], N, \{\#\}, 0, C\text{-True})$
 $\lambda(-, N, U, -). ([], N, U, 0, C\text{-True})$
by $\text{unfold-locales auto}$

lemma trail-conv : $\text{trail } (M, N, U, k, D) = M$ **and**
 clauses-conv : $\text{clauses } (M, N, U, k, D) = N$ **and**
 learned-clss-conv : $\text{learned-clss } (M, N, U, k, D) = U$ **and**
 conflicting-conv : $\text{conflicting } (M, N, U, k, D) = D$ **and**
 $\text{backtrack-lvl-conv}$: $\text{backtrack-lvl } (M, N, U, k, D) = k$

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_W-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_W.clauses-def[simp]

lemma cdcl_W.state-eq-equality[iff]: $cdcl_W.state\text{-}eq \ S \ T \longleftrightarrow S = T$
 unfolding cdcl_W.state-eq-def by (cases S, cases T) auto
declare cdcl_W.state-simp[simp del]

18.3 CDCL Implementation

18.3.1 Definition of the rules

Types **lemma** true-clss-remdups[simp]:
 $I \models s \ (mset \circ \text{remdups}) \ 'N \longleftrightarrow I \models s \ mset \ 'N$
 by (simp add: true-clss-def)

lemma satisfiable-mset-remdups[simp]:
 $\text{satisfiable} \ ((mset \circ \text{remdups}) \ 'N) \longleftrightarrow \text{satisfiable} \ (mset \ 'N)$
 unfolding satisfiable-carac[symmetric] by simp

declare mset-map[symmetric, simp]

value backtrack-split [Marked (Pos (Suc 0)) Level]
value $\exists C \in \text{set} \ [[Pos (Suc 0), Neg (Suc 0)]]. (\forall c \in \text{set } C. -c \in \text{lits-of} \ [\text{Marked} (Pos (Suc 0)) \ \text{Level}])$

type-synonym cdcl_W.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_W-state* and the concrete state *cdcl_W-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 \implies ($\exists C'. z = \text{Propagated } L \ C' \wedge C = \text{mset } C'$)
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_W-state-inv-st* \Rightarrow *nat cdcl_W-state* **where**
toS (M, N, U, k, C) = (map convert M, mset (map mset N), mset (map mset U), k, convertC C)

Definition an abstract type

typedef *cdcl_W-state-inv* = {*S*::*cdcl_W-state-inv-st. cdcl_W-all-struct-inv (toS S)*}
morphisms *rough-state-of state-of*
proof
show ($\square, \square, \square, 0, C\text{-True}$) \in {*S. cdcl_W-all-struct-inv (toS S)*}
by (*auto simp add: cdcl_W-all-struct-inv-def*)
qed

instantiation *cdcl_W-state-inv* :: *equal*

begin

definition *equal-cdcl_W-state-inv* :: *cdcl_W-state-inv* \Rightarrow *cdcl_W-state-inv* \Rightarrow *bool* **where**
equal-cdcl_W-state-inv S S' = (rough-state-of S = rough-state-of S')

instance

by *standard (simp add: rough-state-of-inject equal-cdcl_W-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*

Some (*L*, *C*) \Rightarrow (*Propagated L C # M*, *N*, *U*, *k*, *C-True*)

| *None* \Rightarrow (*M*, *N*, *U*, *k*, *C-True*))

| *S* \Rightarrow *S*)

lemma *do-propagate-step*:

do-propagate-step S $\neq S \implies$ *propagate* (*toS S*) (*toS* (*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]:

conflicting S $\neq C-True \implies$ *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*: $\forall c \in \text{set } (\text{clauses } S @ \text{learned-clss } S).$ *distinct c* **and**

prop-step: *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**

toSS: *toS S* = (*M*, *N*, *U*, *k*, *C-True*) **and**

T: *T* = (*Propagated L (C + {#L#}) # M*, *N*, *U*, *k*, *C-True*) **and**

MC: *M* \models_{as} *CNot 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*: *toS S* = *toS* (*?M*, *?N*, *?U*, *?k*, *?D*)

unfolding *S[symmetric]* **by** *simp*

```

have
  M: M = map convert ?M and
  N: N = mset (map mset ?N) and
  U: U = mset (map mset ?U)
  using toSS[unfolded S] by auto

obtain D where
  DCL: mset D = C + {#L#} and
  D: D ∈ set (?N @ ?U)
  using CL unfolding N U by auto
obtain C' L' where
  setD: set D = set (L' # C') and
  C': 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 ≠ None
  apply (rule dist find-first-unit-clause-none[of D ?N @ ?U ?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 (∀ c ∈ set N. ¬c ∈ lits-of M) then Some N else find-conflict M Ns)

lemma find-conflict-Some:
  find-conflict M Ns = Some N ⟹ N ∈ set Ns ∧ M ⊨as CNot (mset N)
  by (induction Ns rule: find-conflict.induct)
  (auto split: split-if-asm)

lemma find-conflict-None:
  find-conflict M Ns = None ⟷ (∀ N ∈ set Ns. ¬M ⊨as CNot (mset N))
  by (induction Ns) auto

lemma find-conflict-None-no-conflict:
  find-conflict M (N@U) = None ⟷ 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-True) ⇒
        (case find-conflict M (N @ U) of
          Some a ⇒ (M, N, U, k, C-Clause a)
        | None ⇒ (M, N, U, k, C-True))
    | S ⇒ S)

lemma do-conflict-step:
  do-conflict-step S ≠ S ⟹ conflict (toS S) (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  $S = S \implies$  no-step conflict (toS S)
  apply (cases S, cases conflicting S)
  unfolding do-conflict-step-def
  using find-conflict-None-no-confl[of trail S clauses S learned-clss S
    backtrack-lvl S]
  by (auto split: option.splits)

lemma do-conflict-step-conflicting-clause[simp]:
  conflicting  $S \neq C\text{-True} \implies$  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  $S \neq S \implies$  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 o do-conflict-step) S

lemma cp-step-is-cdclW-cp:
  assumes H: do-cp-step  $S \neq S$ 
  shows cdclW-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 cdclW-cp.intros(2)[of ?S ?T] confl unfolding do-cp-step-def by auto
      qed
    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_W-cp-iff-no-propagate-no-conflict:
 no-step cdcl_W-cp $S \longleftrightarrow$ no-step propagate $S \wedge$ no-step conflict S
 by (auto simp: cdcl_W-cp.simps)

lemma do-cp-step-eq-no-step:
 assumes H : do-cp-step $S = S$ and $\forall c \in \text{set } (\text{clauses } S @ \text{learned-clss } S)$. distinct c
 shows no-step cdcl_W-cp (toS S)
 unfolding no-cdcl_W-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-conf[simplified] do-conflict-step-no-step
 split: option.splits)

lemma cdcl_W-cp-cdcl_W-st: cdcl_W-cp $S S' \implies$ cdcl_W** $S S'$
 by (simp add: cdcl_W-cp-tranclp-cdcl_W tranclp-into-rtranclp)

lemma cdcl_W-cp-wf-all-inv: wf $\{(S', S::'v::\text{linorder } \text{cdcl}_W\text{-state}). \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-cp } S S'\}$

(is wf ?R)

proof (rule wf-bounded-measure[of - λS . card (atms-of-msu (clauses S))+1
 λS . length (trail S) + (if conflicting $S = C\text{-True}$ then 0 else 1)], goal-cases)
 case (1 $S S'$)
 then have cdcl_W-all-struct-inv S and cdcl_W-cp $S S'$ by auto
 moreover then have cdcl_W-all-struct-inv S'
 using rtranclp-cdcl_W-all-struct-inv-inv cdcl_W-cp-cdcl_W-st by blast
 ultimately show ?case
 by (auto simp add: cdcl_W-cp.simps elim!: conflictE propagateE
 dest: length-model-le-vars-all-inv)

qed

lemma cdcl_W-all-struct-inv-rough-state[simp]: cdcl_W-all-struct-inv (toS (rough-state-of S))
 using rough-state-of by auto

lemma [simp]: cdcl_W-all-struct-inv (toS S) \implies 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_W-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_W-cp[of rough-state-of S]
 cdcl_W-all-struct-inv-rough-state[of S] cdcl_W-cp-cdcl_W-st rtranclp-cdcl_W-all-struct-inv-inv by blast
 then show ?thesis by auto

qed

Skip fun do-skip-step :: cdcl_W-state-inv-st \Rightarrow cdcl_W-state-inv-st **where**

do-skip-step (Propagated $L C \# Ls, N, U, k$, $C\text{-Clause } D$) =

(if $-L \notin \text{set } D \wedge D \neq []$

then (Ls, N, U, k , $C\text{-Clause } D$)

else (Propagated $L C \# Ls, N, U, k$, $C\text{-Clause } D$)) |

do-skip-step $S = S$

lemma *do-skip-step*:

do-skip-step $S \neq S \implies \text{skip } (\text{toS } S) (\text{toS } (\text{do-skip-step } S))$

apply (*induction* S *rule*: *do-skip-step.induct*)

by (*auto simp add*: *skip.simps*)

lemma *do-skip-step-no*:

do-skip-step $S = S \implies \text{no-step skip } (\text{toS } S)$

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 $S = (a, b, c, d, C\text{-True}) \longleftrightarrow S = (a, b, c, d, C\text{-True})$

by (*cases* S *rule*: *do-skip-step.cases*) *auto*

Resolve fun *maximum-level-code*:: '*a* literal list \Rightarrow ('*a*, nat, '*a* literal list) marked-lit list \Rightarrow nat **where**

maximum-level-code [] = 0 |

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*]:

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* :: *cdcl_W-state-inv-st* \Rightarrow *cdcl_W-state-inv-st* **where**

do-resolve-step (*Propagated* $L \ C \ \# \ Ls, N, U, k, C\text{-Clause } D$) =

(*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 ($Ls, N, U, k, C\text{-Clause } (\text{remdups } (\text{remove1 } L \ C \ @ \ \text{remove1 } (-L) \ D)))$

else (*Propagated* $L \ C \ \# \ Ls, N, U, k, C\text{-Clause } D$)) |

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*:

cdcl_W-all-struct-inv (*toS* S) $\implies \text{do-resolve-step } S \neq S$

$\implies \text{resolve } (\text{toS } S) (\text{toS } (\text{do-resolve-step } S))$

proof (*induction* S *rule*: *do-resolve-step.induct*)

case (1 $L \ C \ M \ N \ U \ k \ D$)

moreover

{ **assume** [*simp*]: $k = 0$

have *get-all-levels-of-marked* (*Propagated* $L \ C \ \# \ M$) = []

using 1(1) **unfolding** *cdcl_W-all-struct-inv-def cdcl_W-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 cdclW-all-struct-inv-def cdclW-conflicting-def by fast
  then have L ∈ set C by fastforce
  then obtain C' where C: mset C = C' + {#L#}
    by (metis add.commute in-multiset-in-set insert-DiffM)
  obtain D' where D: mset D = D' + {#-L#}
    using ⟨- L ∈ set D⟩ 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: mset C - {#L#} + {#L#} = mset C using ⟨L ∈ set C⟩ 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-remdups-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  $\implies$  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]:

```

cdcl_W-all-struct-inv (toS S) \implies *rough-state-of* (state-of (do-resolve-step S)) = do-resolve-step S
apply (rule state-of-inverse)
by (smt CollectI bj *cdcl_W-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) \longleftrightarrow 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) \Rightarrow if i = k \wedge j < i then Some (L, j) else *find-level-decomp* M Ls (L#D) k
)

lemma *find-level-decomp-some*:
assumes *find-level-decomp* M Ls D k = Some (L, j)
shows L \in set Ls \wedge get-maximum-level (mset (remove1 L (Ls @ D))) M = j \wedge 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 \neg (L \in set Ls \wedge get-maximum-level (mset D) M < k \wedge 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'#}) \implies 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' \implies \exists K M2 M1. M = M2 @ M' \wedge 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' \implies *bt-cut* i M \neq None
by (induction M2 arbitrary: M rule: marked-lit-list-induct) auto

lemma *get-all-marked-decomposition-ex*:

$\exists N. (\text{Marked } K (\text{Suc } i) \# M', N) \in \text{set } (\text{get-all-marked-decomposition } (M2 @ \text{Marked } K (\text{Suc } i) \# M'))$

apply (*induction* $M2$ *rule*: *marked-lit-list-induct*)

apply *auto*[2]

by (*case-tac* *get-all-marked-decomposition* ($xs @ \text{Marked } K (\text{Suc } i) \# M'$) *auto*)

lemma *bt-cut-in-get-all-marked-decomposition*:

$\text{bt-cut } i \ M = \text{Some } M' \implies \exists M2. (M', M2) \in \text{set } (\text{get-all-marked-decomposition } M)$

by (*auto* *dest!*: *bt-cut-some-decomp simp add: get-all-marked-decomposition-ex*)

fun *do-backtrack-step* **where**

do-backtrack-step ($M, N, U, k, C\text{-Clause } D$) =

(*case* *find-level-decomp* $M \ D \ [] \ k \ \text{of}$

None $\Rightarrow (M, N, U, k, C\text{-Clause } D)$

| *Some* (L, j) \Rightarrow

(*case* *bt-cut* $j \ M \ \text{of}$

Some ($\text{Marked } - \ - \# Ls$) $\Rightarrow (\text{Propagated } L \ D \# Ls, N, D \# U, j, C\text{-True})$

| $- \Rightarrow (M, N, U, k, C\text{-Clause } D)$)

) |

do-backtrack-step $S = S$

lemma *get-all-marked-decomposition-map-convert*:

(*get-all-marked-decomposition* (*map* *convert* M)) =

map ($\lambda(a, b). (\text{map } \text{convert } a, \text{map } \text{convert } b))$ (*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* $S \neq S$

and *inv*: *cdcl_W-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** $S = \text{this}(1)$ **and** *confl* = *this*(2)

have $E: E = C\text{-Clause } C$ **using** $S \ \text{confl}$ **by** *auto*

obtain $L \ j$ **where** *fd*: *find-level-decomp* $M \ C \ [] \ k = \text{Some } (L, j)$

using *db* **unfolding** $S \ E$ **by** (*cases* C) (*auto split: split-if-asm option.splits*)

have $L \in \text{set } C$ **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** $C: \text{mset } C = \text{mset } C' + \{\#L\# \}$

using ($L \in \text{set } C$) **by** (*metis* *add commute ex-mset in-multiset-in-set insert-DiffM*)

obtain M_2 **where** $M_2: \text{bt-cut } j \ M = \text{Some } M_2$

using *db* *fd* **unfolding** $S \ E$ **by** (*auto split: option.splits*)

obtain $M1 \ K$ **where** $M1: M_2 = \text{Marked } K (\text{Suc } j) \# M1$

using *bt-cut-some-decomp*[*OF* M_2] **by** (*cases* M_2) *auto*

obtain c **where** $c: M = c @ \text{Marked } K (\text{Suc } j) \# M1$

using *bt-cut-in-get-all-marked-decomposition*[*OF* M_2]

unfolding $M1$ **by** *fastforce*

have *get-all-levels-of-marked* (*map* *convert* M) = *rev* [$1..<\text{Suc } k$]

```

    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def S by auto
  from arg-cong[OF this, of  $\lambda a. \text{Suc } j \in \text{set } a$ ] have  $j \leq k$  unfolding c by auto
  have max-l-j: maximum-level-code  $C' M = j$ 
    using db fd  $M_2 C$  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)  $M \geq k$ 
    using  $\langle L \in \text{set } C \rangle$  get-maximum-level-ge-get-level levL by blast
  moreover have get-maximum-level (mset C)  $M \leq k$ 
    using get-maximum-level-exists-lit-of-max-level[of mset C M] inv
      cdclW-M-level-inv-get-level-le-backtrack-lvl[of toS S]
    unfolding C cdclW-all-struct-inv-def S
    by auto metis+
  ultimately have get-maximum-level (mset C)  $M = k$  by auto

  obtain M2 where  $M2: (M_2, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
    using bt-cut-in-get-all-marked-decomposition[OF M2] 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),  $\{\#mset C' + \{\#L\#\}\# \} + \text{mset } (\text{map mset } U), j, C-True)$ 
    apply (subst state-conv[of cdclW.reduce-trail-to - -])
    using M2 unfolding M1 by auto
  have
    backtrack
      (map convert M, mset  $\# \text{ mset } N$ , mset  $\# \text{ mset } U$ , k, C-Clause (mset C))
      (Propagated L (mset C)  $\# \text{ map convert } M1$ , mset  $\# \text{ mset } N$ , mset  $\# \text{ mset } U + \{\#mset C\# \}$ ,
    j,
      C-True)
  apply (rule backtrack-rule)
    unfolding C apply simp
    using Set.imageI[of  $(M_2, M2)$  set (get-all-marked-decomposition M)
       $(\lambda(a, b). (\text{map convert } a, \text{map convert } b))$ ] M2
    apply (auto simp: get-all-marked-decomposition-map-convert M1)[1]
    using max-l-j levL  $\langle j \leq k \rangle$  apply (simp add: get-maximum-level-plus)
    using C  $\langle \text{get-maximum-level (mset C) } M = k \rangle$  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  $(M_2, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
    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  $\bigwedge p. \neg \text{cdcl}_W\text{-o } (toS\ S) p \vee \text{cdcl}_W\text{-all-struct-inv } p$ 
    using inv cdclW-all-struct-inv-inv other by blast
  then have do-backtrack-step  $S = S \vee \text{cdcl}_W\text{-all-struct-inv } (toS\ (do\text{-backtrack-step } S))$ 
    using f2 by blast
  then show do-backtrack-step  $S \in \{S. \text{cdcl}_W\text{-all-struct-inv } (toS\ S)\}$ 
    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  $\Rightarrow$  (M, N, U, k, C-True)
  | Some L  $\Rightarrow$  (Marked L (Suc k) # M, N, U, k+1, C-True)) |
do-decide-step S = S

```

lemma do-decide-step:

```

do-decide-step  $S \neq S \implies \text{decide } (toS\ S) (toS\ (do\text{-decide-step } S))$ 
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-ms-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  $\bigwedge f. (f\ m :: \text{nat literal multiset}) \in f\ ' \text{set } b$ 
    using a1 by blast
  then have  $\bigwedge f. (\text{atms-of } (f\ m) :: \text{nat set}) \subseteq \text{atms-of-ms } (f\ ' \text{set } b)$ 
    using atms-of-atms-of-ms-mono by blast
  then have  $\bigwedge n f. (n :: \text{nat}) \in \text{atms-of-ms } (f\ ' \text{set } b) \vee n \notin \text{atms-of } (f\ m)$ 
    by (meson contra-subsetD)
  then have  $\text{atm-of } x2 \in \text{atms-of-ms } (\text{mset } ' \text{set } b)$ 
    using f2 by blast
} note H = this
assume do-decide-step  $S \neq S$  and
  S = (a, b, c, d, e) and
  conflicting S = C-True
then show decide (toS S) (toS (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  $S = S \implies \text{no-step decide } (toS\ S)$ 

```

```

apply (cases  $S$ , cases conflicting  $S$ )
apply (auto
  simp add: atms-of-ms-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]:
   $cdcl_W\text{-all-struct-inv } (toS\ S) \implies \text{rough-state-of } (state\text{-of } (do\text{-decide-step } S)) = do\text{-decide-step } S$ 
apply (subst state-of-inverse)
apply (smt  $cdcl_W\text{-all-struct-inv-inv decide do-decide-step mem-Collect-eq other}$ )
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  $cdcl_W\text{-all-struct-inv-inv skip do-skip-step mem-Collect-eq other bj}$ )
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  $xs = state\text{-of } (if\ cdcl_W\text{-all-struct-inv } (toS\ (fst\ xs, snd\ xs))\ then\ xs$ 
  else  $([], [], [], 0, C\text{-True}))$ 

```

```

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'  $S = state\text{-of } (do\text{-cp-step } (rough\text{-state-of } S))$ 

```

```

typedef  $cdcl_W\text{-state-inv-from-init-state} = \{S::cdcl_W\text{-state-inv-st. } cdcl_W\text{-all-struct-inv } (toS\ S)$ 
   $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W\ (clauses\ (toS\ S))) (toS\ S)\}$ 
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:  $cdcl_W\text{-all-struct-inv-def}$ )
qed

```

```

instantiation  $cdcl_W\text{-state-inv-from-init-state} :: equal$ 
begin

```

```

definition  $equal\text{-}cdcl_W\text{-state-inv-from-init-state} :: cdcl_W\text{-state-inv-from-init-state} \Rightarrow$ 
   $cdcl_W\text{-state-inv-from-init-state} \Rightarrow bool$  where
   $equal\text{-}cdcl_W\text{-state-inv-from-init-state } S\ S' \longleftrightarrow$ 
   $(rough\text{-state-from-init-state-of } S = rough\text{-state-from-init-state-of } S')$ 

```


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-stgy*** (*S0-cdcl_W* (*clauses* (*toS S*))) (*toS S*) *then S else* (\square , \square , \square , \emptyset , *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_W-state-inv* \Rightarrow *cdcl_W-state-inv* **where**

do-full1-cp-step S =

(*let S' = do-cp-step' S in*

if S = S' then S else do-full1-cp-step S')

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_W-cp-wf-all-inv, of toS*], *of rough-state-of*] .

next

case ($\exists S' S$)

then show *?case*

unfolding *do-cp-step'-def*

apply *simp*

by (*metis cp-step-is-cdcl_W-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_W-all-struct-inv-def*

distinct-cdcl_W-state-def)

lemma *do-full1-cp-step-full:*

full cdcl_W-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_W-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 $U \neq T$
 then U *else*
 (*let* V = *do-backtrack-step* U *in*
 if $V \neq U$ *then* V *else* *do-decide-step* V)))

lemma *do-other-step*:
assumes *inv*: *cdcl_W-all-struct-inv* (*toS* S) **and**
st: *do-other-step* $S \neq S$
shows *cdcl_W-o* (*toS* S) (*toS* (*do-other-step* S))
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_W-all-struct-inv* (*toS* S) **and**
st: *do-other-step* $S = S$
shows *no-step cdcl_W-o* (*toS* S)
using *st inv* **by** (*auto split: split-if-asm elim: cdcl_W-bjE*
simp add: Let-def cdcl_W-bj.simps elim!: cdcl_W-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_W-o* (*toS* (*rough-state-of* S)) (*toS* (*do-other-step* (*rough-state-of* S)))
 by (*metis* *False cdcl_W-all-struct-inv-rough-state do-other-step*[*of* *rough-state-of* S])
 then have *cdcl_W-all-struct-inv* (*toS* (*do-other-step* (*rough-state-of* S)))
 using *cdcl_W-all-struct-inv-inv cdcl_W-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_W-all-struct-inv-inv*
cdcl_W-all-struct-inv-rough-state *mem-Collect-eq* *other* *state-of-inverse*)

definition *do-cdcl_W-stgy-step* **where**

do-cdcl_W-stgy-step $S =$
 (*let* $T =$ *do-full1-cp-step* S *in*
 if $T \neq S$
 then T
 else
 (*let* $U =$ (*do-other-step'* T) *in*
 (*do-full1-cp-step* U)))

definition *do-cdcl_W-stgy-step'* **where**

do-cdcl_W-stgy-step' $S =$ *state-from-init-state-of* (*rough-state-of* (*do-cdcl_W-stgy-step* (*id-of-I-to* S)))

lemma *toS-do-full1-cp-step-not-eq*: *do-full1-cp-step* $S \neq S \implies$

toS (*rough-state-of* S) \neq *toS* (*rough-state-of* (*do-full1-cp-step* S))

proof –

assume *a1*: *do-full1-cp-step* $S \neq S$

then have $S \neq$ *do-cp-step'* S

by *fastforce*

then show *?thesis*

by (*metis* (*no-types*) *cp-step-is-cdcl_W-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

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps* [*simp del*]

Correction of the transformation **lemma** *do-cdcl_W-stgy-step*:

assumes *do-cdcl_W-stgy-step* $S \neq S$

shows *cdcl_W-stgy* (*toS* (*rough-state-of* S)) (*toS* (*rough-state-of* (*do-cdcl_W-stgy-step* S)))

proof (*cases* *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_W-stgy-step-def*

by (*auto* *intro!*: *cdcl_W-stgy.intros* *dest*: *toS-do-full1-cp-step-not-eq*)

next

case *True*

have *cdcl_W-o* (*toS* (*rough-state-of* S)) (*toS* (*rough-state-of* (*do-other-step'* S)))

by (*smt* *True* *assms* *cdcl_W-all-struct-inv-rough-state* *do-cdcl_W-stgy-step-def* *do-other-step*
rough-state-of-do-other-step' *rough-state-of-inverse*)

moreover

have

np: *no-step* *propagate* (*toS* (*rough-state-of* S)) **and**

nc: *no-step* *conflict* (*toS* (*rough-state-of* S))

```

    apply (metis True do-cp-step-eq-no-prop-no-conf
      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-conf
    do-full1-cp-step-fix-point-of-do-full1-cp-step)
then have no-step cdclW-cp (toS (rough-state-of S))
  by (simp add: cdclW-cp.simps)
moreover have full cdclW-cp (toS (rough-state-of (do-other-step' S)))
  (toS (rough-state-of (do-full1-cp-step (do-other-step' S))))
  using do-full1-cp-step-full by auto
ultimately show ?thesis
  using assms True unfolding do-cdclW-stgy-step-def
  by (auto intro!: cdclW-stgy.other' dest: toS-do-full1-cp-step-not-eq)
qed

```

```

lemma length-trail-toS[simp]:
  length (trail (toS S)) = length (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-neq-imp-trail-neq:
  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  $\langle \text{cdcl}_W\text{-M-level-inv } (toS S) \rangle$ 
    proof (induction toS (do-other-step S) rule: cdclW-o-induct-lev2)
      case decide
      then show ?thesis
        by (auto simp add: trail-toS-neq-imp-trail-neq)[]
    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-clss 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 $S = C\text{-True}$ **and**
do-decide-step $S \neq S$
shows $\text{Suc } (\text{length } (\text{filter is-marked } (\text{trail } S)))$
 $= \text{length } (\text{filter is-marked } (\text{trail } (\text{do-decide-step } S)))$
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* $S \neq C\text{-True}$ **and**
do-decide-step $S \neq S$
shows $\text{length } (\text{filter is-marked } (\text{trail } S)) < \text{length } (\text{filter is-marked } (\text{trail } (\text{do-decide-step } S)))$
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 $\text{length } (\text{filter is-marked } (\text{trail } (\text{rough-state-of } S)))$
 $> \text{length } (\text{filter is-marked } (\text{trail } (\text{rough-state-of } (\text{do-other-step'} S))))$
proof (*cases* S , *goal-cases*)
case ($1\ y$) **note** $S = \text{this}(1)$ **and** $\text{inv} = \text{this}(2)$
obtain $M\ N\ U\ k\ E$ **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'* $S = \text{state-of } (\text{do-backtrack-step } (\text{rough-state-of } S))$

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* ($(M, N, U, k, C\text{-True})$)) $\neq \text{state-of } (M, N, U, k, C\text{-True})$

```

    using assms(2) unfolding do-other-step'-def y 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) ≠ 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]:*
 $\text{rough-state-of } (\text{state-of } (\text{do-skip-step } (\text{rough-state-of } S))) = \text{do-skip-step } (\text{rough-state-of } S)$
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*:

trail (*do-other-step* *S*) = *trail* *S* \longleftrightarrow *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*: *do-full1-cp-step* (*do-other-step'* *S*) = *S*

shows *do-other-step'* *S* = *S* \wedge *do-full1-cp-step* *S* = *S*

proof –

let *?T* = *do-other-step'* *S*

{ **assume** *confl*: *conflicting* (*rough-state-of* *?T*) \neq *C-True*

then have *tr*: *trail* (*rough-state-of* (*do-full1-cp-step* *?T*)) = *trail* (*rough-state-of* *?T*)

using *do-full1-cp-step-conflicting* **by** *auto*

have *trail* (*rough-state-of* (*do-full1-cp-step* (*do-other-step'* *S*))) = *trail* (*rough-state-of* *S*)

using *arg-cong[OF H, of $\lambda S. \text{trail} (\text{rough-state-of } S)$]* .

then have *trail* (*rough-state-of* (*do-other-step'* *S*)) = *trail* (*rough-state-of* *S*)

by (*auto simp add*: *do-full1-cp-step-conflicting confl*)

then have *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]*: *do-other-step'* *S* = *S*

obtain *c* **where** *c*: *trail* (*rough-state-of* (*do-full1-cp-step* *S*)) = *c* @ *trail* (*rough-state-of* *S*)

using *do-full1-cp-step-neq-trail-increase* **by** *auto*

moreover have *trail* (*rough-state-of* (*do-full1-cp-step* *S*)) = *trail* (*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 *do-full1-cp-step* *S* = *S* **using** *assms* **by** *auto*

}

moreover {

assume *confl*: *conflicting* (*rough-state-of* *?T*) = *C-True* **and** *neq*: *do-other-step'* *S* \neq *S*

obtain *c* **where**

c: *trail* (*rough-state-of* (*do-full1-cp-step* *?T*)) = *c* @ *trail* (*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 *length* (*filter is-marked* (*trail* (*rough-state-of* (*do-full1-cp-step* *?T*))))

= *length* (*filter is-marked* (*trail* (*rough-state-of* *?T*))) **using** *nm* **unfolding** *c* **by** *force*

moreover have *length* (*filter is-marked* (*trail* (*rough-state-of* *S*)))

\neq *length* (*filter is-marked* (*trail* (*rough-state-of* *?T*)))

using *do-other-step-not-conflicting-one-more-decide[OF - neq]*

do-other-step-not-conflicting-one-more-decide-bt[of S, OF - confl neq]

by *linarith*

finally have *False* **unfolding** *H* **by** *blast*

}

ultimately show *?thesis* **by** *blast*

qed

lemma *do-cdcl_W-stgy-step-no*:

assumes *S*: *do-cdcl_W-stgy-step* *S* = *S*

shows *no-step cdcl_W-stgy* (*toS* (*rough-state-of* *S*))


```

proof -
{
  fix S'
  assume full1 cdclW-cp (toS (rough-state-of S)) S'
  then have False
    using do-full1-cp-step-full[of S] unfolding full-def S rtrancpl-unfold full1-def
    by (smt assms do-cdclW-stgy-step-def trancplD)
}
moreover {
  fix S' S''
  assume cdclW-o (toS (rough-state-of S)) S' and
    no-step propagate (toS (rough-state-of S)) and
    no-step conflict (toS (rough-state-of S)) and
    full cdclW-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-rtrancpl-cdclW: cdclW-cp S T  $\implies$  cdclW** S T
  apply (induction rule: cdclW-cp.induct)
  using conflict apply blast
  using propagate by blast

lemma rtrancpl-cdclW-cp-is-rtrancpl-cdclW: cdclW-cp** S T  $\implies$  cdclW** S T
  apply (induction rule: rtrancpl-induct)
  apply simp
  by (fastforce dest!: cdclW-cp-is-rtrancpl-cdclW)

lemma cdclW-stgy-is-rtrancpl-cdclW:
  cdclW-stgy S T  $\implies$  cdclW** S T
  apply (induction rule: cdclW-stgy.induct)
  using cdclW-stgy.conflict' rtrancpl-cdclW-stgy-rtrancpl-cdclW apply blast
  unfolding full-def by (fastforce dest!: cdclW.other rtrancpl-cdclW-cp-is-rtrancpl-cdclW)

lemma cdclW-stgy-init-clss: cdclW-stgy S T  $\implies$  cdclW-M-level-inv S  $\implies$  clauses S = clauses T
  using rtrancpl-cdclW-init-clss cdclW-stgy-is-rtrancpl-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]:

```

$\text{rough-state-from-init-state-of } (\text{do-cdcl}_W\text{-stgy-step}' S) =$
 $\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{id-of-I-to } S))$
proof –
let $?S = (\text{rough-state-from-init-state-of } S)$
have $\text{cdcl}_W\text{-stgy}^{**} (S0\text{-cdcl}_W (\text{clauses } (\text{toS } (\text{rough-state-from-init-state-of } S))))$
 $(\text{toS } (\text{rough-state-from-init-state-of } S))$
using $\text{rough-state-from-init-state-of}[of\ S]$ **by** *auto*
moreover have $\text{cdcl}_W\text{-stgy}^{**}$
 $(\text{toS } (\text{rough-state-from-init-state-of } S))$
 $(\text{toS } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step}$
 $(\text{state-of } (\text{rough-state-from-init-state-of } S))))))$
using $\text{do-cdcl}_W\text{-stgy-step}[of\ \text{state-of } ?S]$
by $(\text{cases } \text{do-cdcl}_W\text{-stgy-step } (\text{state-of } ?S) = \text{state-of } ?S)$ *auto*
ultimately show *?thesis*
unfolding $\text{do-cdcl}_W\text{-stgy-step}'\text{-def id-of-I-to-def}$ **by** $(\text{auto intro!}:\ \text{state-from-init-state-of-inverse})$
qed

All rules together function $\text{do-all-cdcl}_W\text{-stgy}$ **where**

$\text{do-all-cdcl}_W\text{-stgy } S =$
 $(\text{let } T = \text{do-cdcl}_W\text{-stgy-step}' S \text{ in}$
 $\text{if } T = S \text{ then } S \text{ else } \text{do-all-cdcl}_W\text{-stgy } T)$

by *fast+*

termination

proof $(\text{relation } \{(T, S).$

$(\text{cdcl}_W\text{-measure } (\text{toS } (\text{rough-state-from-init-state-of } T))),$
 $\text{cdcl}_W\text{-measure } (\text{toS } (\text{rough-state-from-init-state-of } S)))$
 $\in \text{lexn } \{(a, b). a < b\} \exists\}$, *goal-cases*)

case 1

show *?case* **by** $(\text{rule wf-if-measure-f}) (\text{auto intro!}:\ \text{wf-lexn wf-less})$

next

case $(2\ S\ T)$ **note** $T = \text{this}(1)$ **and** $ST = \text{this}(2)$

let $?S = \text{rough-state-from-init-state-of } S$

have $S: \text{cdcl}_W\text{-stgy}^{**} (S0\text{-cdcl}_W (\text{clauses } (\text{toS } ?S))) (\text{toS } ?S)$

using $\text{rough-state-from-init-state-of}[of\ S]$ **by** *auto*

moreover have $\text{cdcl}_W\text{-stgy } (\text{toS } (\text{rough-state-from-init-state-of } S))$

$(\text{toS } (\text{rough-state-from-init-state-of } T))$

using $ST\ \text{do-cdcl}_W\text{-stgy-step}$ **unfolding** T

by $(\text{smt id-of-I-to-def mem-Collect-eq rough-state-from-init-state-of}$
 $\text{rough-state-from-init-state-of-do-cdcl}_W\text{-stgy-step}' \text{ rough-state-from-init-state-of-inject}$
 $\text{state-of-inverse})$

moreover

have $\text{cdcl}_W\text{-all-struct-inv } (\text{toS } (\text{rough-state-from-init-state-of } S))$

using $\text{rough-state-from-init-state-of}[of\ S]$ **by** *auto*

then have $\text{cdcl}_W\text{-all-struct-inv } (S0\text{-cdcl}_W (\text{clauses } (\text{toS } (\text{rough-state-from-init-state-of } S))))$

by $(\text{cases } \text{rough-state-from-init-state-of } S)$

$(\text{auto simp add: } \text{cdcl}_W\text{-all-struct-inv-def distinct-cdcl}_W\text{-state-def})$

ultimately show *?case*

by $(\text{auto intro!}:\ \text{cdcl}_W\text{-stgy-step-decreasing}[of\ -\ S0\text{-cdcl}_W (\text{clauses } (\text{toS } ?S))]$
 $\text{simp del: } \text{cdcl}_W\text{-measure.simps})$

qed

thm $\text{do-all-cdcl}_W\text{-stgy.induct}$

lemma $\text{do-all-cdcl}_W\text{-stgy.induct}:$

$(\bigwedge S. (\text{do-cdcl}_W\text{-stgy-step}' S \neq S \implies P (\text{do-cdcl}_W\text{-stgy-step}' S)) \implies P S) \implies P a0$

using $\text{do-all-cdcl}_W\text{-stgy.induct}$ **by** *metis*

```

lemma no-step-cdclW-stgy-cdclW-all:
  no-step cdclW-stgy (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)
proof –
  fix Sa :: cdclW-state-inv-from-init-state
  assume a1: ¬ do-cdclW-stgy-step' Sa ≠ Sa
  { fix pp
    have (if True then Sa else do-all-cdclW-stgy Sa) = do-all-cdclW-stgy Sa
      using a1 by auto
    then have ¬ cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa))) pp
      using a1 by (metis (no-types) do-cdclW-stgy-step-no id-of-I-to-def
        rough-state-from-init-state-of-do-cdclW-stgy-step' rough-state-of-inverse) }
    then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-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_W-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_W-all-struct-inv (toS ([], map remdups N, [], 0, C-True))*

unfolding *cdcl_W-all-struct-inv-def distinct-cdcl_W-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_W-stgy (toS ([], ?N, [], 0, C-True)) (toS S)*

unfolding *full-def* **apply** *rule*

using *do-all-cdcl_W-stgy-is-rtrancpl-cdcl_W-stgy* [of

```

state-from-init-state-of ([], map remdups N, [], 0, C-True)] inv
no-step-cdclW-stgy-cdclW-all
by (auto simp del: do-all-cdclW-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 cdclW-all-struct-inv (toS S)
  by (metis (no-types) cdclW-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 cdclW-all-struct-inv-def cdclW-M-level-inv-def S[symmetric] by auto
moreover
have clauses (toS ([], ?N, [], 0, C-True)) = clauses (toS S)
  apply (rule rtrancp-cdclW-init-clss)
  using 1 unfolding full-def by (auto simp add: rtrancp-cdclW-stgy-rtrancp-cdclW)
then have N': mset (map mset ?N) = N'
  using S[symmetric] by auto
have (E ≠ C-Clause {#} ∧ satisfiable (set (map mset ?N)))
  ∨ (E = C-Clause {#} ∧ 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-cl)
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' ⇒ cdclW-M-level-inv S ⇒
  (∧ D L K M1 M2.
    (Marked K (Suc (get-maximum-level D (trail S))) # M1, M2)
    ∈ set (get-all-marked-decomposition (trail S)) ⇒
    get-level L (trail S) = get-maximum-level (D + {#L#}) (trail S) ⇒
    undefined-lit M1 L ⇒

```

$S' \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\# \}))$
 $(\text{reduce-trail-to } M1 \ (\text{add-learned-cls } (D + \{\#L\# \}))$
 $(\text{update-backtrack-lvl } (\text{get-maximum-level } D \ (\text{trail } S)) \ (\text{update-conflicting } C\text{-True } S))) \implies$
 $\text{backtrack-lvl } S = \text{get-maximum-level } (D + \{\#L\# \}) \ (\text{trail } S) \implies$
 $\text{conflicting } S = C\text{-Clause } (D + \{\#L\# \}) \implies P \implies$
 P

using *assms* **by** (*induction rule: backtrack-induction-lev2*) *metis*

lemma *backtrack-no-cdcl_W-bj*:

assumes *cdcl*: *cdcl_W-bj* *T U* **and** *inv*: *cdcl_W-M-level-inv* *V*

shows $\neg \text{backtrack } V \ T$

using *cdcl inv*

apply (*induction rule: cdcl_W-bj.induct*)

apply (*elim skipE*, *force elim!*: *backtrack-levE*[*OF - inv*] *simp: cdcl_W-M-level-inv-def*)

apply (*elim resolveE*, *force elim!*: *backtrack-levE*[*OF - inv*] *simp: cdcl_W-M-level-inv-def*)

apply *standard*

apply (*elim backtrack-levE*[*OF - inv*], *elim backtrackE*)

apply (*force simp del: state-simp simp add: state-eq-conflicting cdcl_W-M-level-inv-decomp*)

done

abbreviation *skip-or-resolve* :: *'st* \Rightarrow *'st* \Rightarrow *bool* **where**

skip-or-resolve $\equiv (\lambda S \ T. \text{skip } S \ T \vee \text{resolve } S \ T)$

lemma *rtranclp-cdcl_W-bj-skip-or-resolve-backtrack*:

assumes *cdcl_W-bj*** *S U* **and** *inv*: *cdcl_W-M-level-inv* *S*

shows *skip-or-resolve*** *S U* $\vee (\exists T. \text{skip-or-resolve** } S \ T \wedge \text{backtrack } T \ U)$

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_W-bj⁺⁺* *S U*

using *st* **unfolding** *rtranclp-unfold* **by** *blast*

then show *?case*

proof *cases*

case *SUp*

have $\bigwedge T. \text{skip-or-resolve** } S \ T \implies \text{cdcl}_W^{**} \ S \ T$

using *mono-rtranclp*[*of skip-or-resolve cdcl_W*] *other* **by** *blast*

then have *skip-or-resolve*** *S U*

using *bj IH inv backtrack-no-cdcl_W-bj rtranclp-cdcl_W-consistent-inv*[*OF - inv*] **by** *meson*

then show *?thesis*

using *bj* **by** (*metis* (*no-types, lifting*) *cdcl_W-bj.cases rtranclp.simps*)

next

case *SU*

then show *?thesis*

using *bj* **by** (*metis* (*no-types, lifting*) *cdcl_W-bj.cases rtranclp.simps*)

qed

qed

lemma *rtranclp-skip-or-resolve-rtranclp-cdcl_W*:

*skip-or-resolve*** *S T* $\implies \text{cdcl}_W^{**} \ S \ T$

by (*induction rule: rtracp-induct*) (*auto dest!:* *cdcl_W-bj.intros cdcl_W.intros cdcl_W-o.intros*)

definition *backjump-l-cond* :: '*v clause* \Rightarrow '*v clause* \Rightarrow '*v literal* \Rightarrow '*st* \Rightarrow *bool* **where**
backjump-l-cond $\equiv \lambda C\ C'\ L'\ S.$ *True*

definition *inv_{NOT}* :: '*st* \Rightarrow *bool* **where**
inv_{NOT} $\equiv \lambda S.$ *no-dup* (*trail* *S*)

declare *inv_{NOT}-def*[*simp*]
end

fun *convert-marked-lit-from-W* **where**
convert-marked-lit-from-W (*Propagated* *L* -) = *Propagated* *L* () |
convert-marked-lit-from-W (*Marked* *L* -) = *Marked* *L* ()

abbreviation *convert-trail-from-W* ::
 ('*v*, '*vl*, '*a*) *marked-lit list*
 \Rightarrow ('*v*, *unit*, *unit*) *marked-lit list* **where**
convert-trail-from-W $\equiv \text{map } \text{convert-marked-lit-from-W}$

lemma *atm-convert-trail-from-W*[*simp*]:
 ($\lambda l.$ *atm-of* (*lit-of* *l*)) '*set* (*convert-trail-from-W* *xs*) = ($\lambda l.$ *atm-of* (*lit-of* *l*)) '*set* *xs*
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 *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-marked-lit-from-W* *L*) = *lit-of* *L*
by (*cases* *L*) *auto*

lemma *no-dup-convert-from-W*[*simp*]:
no-dup (*convert-trail-from-W* *M*) \longleftrightarrow *no-dup* *M*
by (*auto simp: comp-def*)

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 image-comp*)

The values *0* and $\{\#\}$ are dummy values.

fun *convert-marked-lit-from-NOT*
 :: ('*a*, '*e*, '*b*) *marked-lit* \Rightarrow ('*a*, *nat*, '*a literal multiset*) *marked-lit* **where**
convert-marked-lit-from-NOT (*Propagated* *L* -) = *Propagated* *L* $\{\#\}$ |
convert-marked-lit-from-NOT (*Marked* *L* -) = *Marked* *L* *0*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT $\equiv \text{map } \text{convert-marked-lit-from-NOT}$

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-convert-lit-from-NOT[simp]:*
convert-marked-lit-from-W (convert-marked-lit-from-NOT L) = L
by (*cases L*) *auto*

abbreviation *trail_{NOT}* **where**
trail_{NOT} S \equiv *convert-trail-from-W (fst S)*

lemma *undefined-lit-convert-trail-from-W[iff]:*
undefined-lit (convert-trail-from-W M) L \longleftrightarrow undefined-lit M L
by (*auto simp: defined-lit-map image-comp*)

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*
 $\lambda S.$ *convert-trail-from-W (trail S)*
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 simp: map-tl o-def)*

context *state_W*
begin
declare *state-simp_{NOT}[simp del]*
end

sublocale *cdcl_W-ops \subseteq cdcl_{NOT}-merge-bj-learn-ops*
 $\lambda S.$ *convert-trail-from-W (trail S)*
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*
 $\lambda C C' L' S.$ *backjump-l-cond C 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*
 $\lambda S.$ *convert-trail-from-W (trail S)*
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 simp: comp-def)*

```

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  $\langle inv_{NOT}\ S \rangle \langle convert-trail-from-W\ (trail\ S) = F' @\ Marked\ K\ () \# F \rangle$ 
    unfolding invNOT-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  $\langle F \models_{as} CNot\ C' \rangle \langle undefined-lit\ F\ L \rangle$  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 (trail S))
    using  $\langle inv_{NOT}\ S \rangle$  unfolding invNOT-def by (simp add: o-def)
  have f3: atm-of L  $\in$  atms-of-msu (clauses S)
     $\cup$  atm-of ' lits-of (convert-trail-from-W (trail S))
    using  $\langle convert-trail-from-W\ (trail\ S) = F' @\ Marked\ K\ () \# F \rangle$ 
     $\langle atm-of\ L \in atms-of-msu\ (clauses\ S) \cup atm-of\ ' lits-of\ (F' @\ Marked\ K\ () \# F) \rangle$  by auto
  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[OF - f2] calculation(2-5,9)
    state-eqNOT-ref unfolding backjump-l-cond-def by blast
qed
qed

sublocale cdclW-ops  $\subseteq$  cdclNOT-merge-bj-learn-proxy2
 $\lambda S.$  convert-trail-from-W (trail S)
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

```



```

λS. convert-trail-from-W (trail S)
clauses
λL S. cons-trail (convert-marked-lit-from-NOT L) S
λS. tl-trail S
λC S. add-learned-cls C S
λC S. remove-cls C S λ- -. True invNOT
λ- S. conflicting S = C-True backjump-l-cond
apply unfold-locales
using dpll-bj-no-dup apply (simp add: comp-def)
using cdclNOT-no-dup by (auto simp add: comp-def cdclNOT.simps)

```

```

context cdclW-ops
begin

```

Notations are lost while proving locale inclusion:

```

notation state-eqNOT (infix ~NOT 50)

```

19.2 Additional Lemmas between NOT and W states

```

lemma trailW-eq-reduce-trail-toNOT-eq:
  trail S = trail T  $\implies$  trail (reduce-trail-toNOT F S) = trail (reduce-trail-toNOT F T)
proof (induction F S arbitrary: T rule: reduce-trail-toNOT.induct)
case (1 F S T) note IH = this(1) and tr = this(2)
then have [] = convert-trail-from-W (trail S)
  ∨ length F = length (convert-trail-from-W (trail S))
  ∨ trail (reduce-trail-toNOT F (tl-trail S)) = trail (reduce-trail-toNOT F (tl-trail T))
using IH by (metis (no-types) trail-tl-trail)
then show trail (reduce-trail-toNOT F S) = trail (reduce-trail-toNOT F T)
using tr by (metis (no-types) reduce-trail-toNOT.elims)
qed

```

```

lemma trail-reduce-trail-toNOT-add-learned-cls[simp]:
  no-dup (trail S)  $\implies$ 
  trail (reduce-trail-toNOT M (add-learned-cls D S)) = trail (reduce-trail-toNOT M S)
by (rule trailW-eq-reduce-trail-toNOT-eq) simp

```

```

lemma reduce-trail-toNOT-reduce-trail-convert:
  reduce-trail-toNOT C S = reduce-trail-to (convert-trail-from-NOT C) S
apply (induction C S rule: reduce-trail-toNOT.induct)
apply (subst reduce-trail-toNOT.simps, subst reduce-trail-to.simps)
by auto

```

```

lemma reduce-trail-to-length:
  length M = length M'  $\implies$  reduce-trail-to M S = reduce-trail-to M' S
apply (induction M S arbitrary: rule: reduce-trail-to.induct)
apply (case-tac trail S  $\neq$  [] ; case-tac length (trail S)  $\neq$  length M'; simp)
by (simp-all add: reduce-trail-to-length-ne)

```

19.3 More lemmas conflict-propagate and backjumping

19.3.1 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_W-bj-measure*:

assumes *cdcl_W-bj S T* **and** *cdcl_W-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_W-bj.induct*)
(force dest:arg-cong[of - - length]
intro: get-all-marked-decomposition-exists-prepend
elim!: backtrack-levE
simp: cdcl_W-M-level-inv-def)+

lemma *wf-cdcl_W-bj*:

wf {(b,a). cdcl_W-bj a b ∧ cdcl_W-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_W-bj-measure* **by** *blast*

lemma *cdcl_W-bj-exists-normal-form*:

assumes *lev: cdcl_W-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_W-bj a b ∧ cdcl_W-M-level-inv a) S T*
using *wf-exists-normal-form-full[OF wf-cdcl_W-bj]* **by** *auto*
then have *cdcl_W-bj** S T*
by (*auto dest: rtrancpl-and-rtrancpl-left simp: full-def*)

moreover

then have *cdcl_W** S T*
using *mono-rtrancpl[of cdcl_W-bj cdcl_W] cdcl_W.simps* **by** *blast*
then have *cdcl_W-M-level-inv T*
using *rtrancpl-cdcl_W-consistent-inv lev* **by** *auto*
ultimately show *?thesis* **using** *T unfolding full-def* **by** *auto*

qed

lemma *rtrancpl-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**
 $T \sim \text{delete-trail-and-rebuild } (\text{trail } T) \ S$
using *assms* **by** (*induction rule: rtrancpl-induct*) (*auto simp del: state-simp simp: state-eq-def*)+

19.3.2 More backjumping

Backjumping after skipping or jump directly **lemma** *rtrancpl-skip-backtrack-backtrack*:

assumes
*skip** S T* **and**
backtrack T W **and**
cdcl_W-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-clss S = N and [simp]: learned-clss 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-clss (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 [skip** ?S ?T; backtrack ?T ?W; cdclW-all-struct-inv ?S] ⇒ 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 $W = (Propagated L ((D + \{\#L\#\})) \# M1, N, \{\#D + \{\#L\#\}\# + U,$
 $get\text{-}maximum\text{-}level D M, C\text{-}True)$ **and**
decomp: $(Marked K (i+1) \# M1, M2) \in set (get\text{-}all\text{-}marked\text{-}decomposition M)$ **and**
lev-l: $get\text{-}level L M = k$ **and**
lev-l-D: $get\text{-}level L M = get\text{-}maximum\text{-}level (D + \{\#L\#\}) M$ **and**
i: $i = get\text{-}maximum\text{-}level D M$ **and**
undef: $undefined\text{-}lit M1 L$
using *bt* **by** $(elim\ backtrack\text{-}levE) (force\ simp: cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def) +$
let $?D = (D + \{\#L\#\})$

have $[simp]: no\text{-}dup (trail S)$
using *M-lev* **by** $(auto\ simp: cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp)$
have $cdcl_W\text{-}all\text{-}struct\text{-}inv T$
using *mono-rtrancp* $[of\ skip\ cdcl_W]$ **by** $(smt\ bj\ cdcl_W\text{-}bj.skip\ inv\ local.skip\ other$
 $rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv)$
then have $[simp]: no\text{-}dup (trail T)$
unfolding $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *auto*

obtain $MS M_T$ **where** $M: M = MS @ M_T$ **and** $M_T: M_T = trail T$ **and** $nm: \forall m \in set\ MS. \neg is\text{-}marked$
 m
using *rtrancp-skip-state-decomp* $(1)[OF\ skip] S\ M\text{-}lev$ **by** *auto*
have *T*: state $T = (M_T, N, U, k, C\text{-}Clause\ ?D)$
using $M_T\ rtrancp\text{-}skip\text{-}state\text{-}decomp(2)[of\ S\ T]\ skip\ S$
by $(auto\ simp\ del: state\text{-}simp\ simp: state\text{-}eq\text{-}def)$

have $cdcl_W\text{-}all\text{-}struct\text{-}inv T$
apply $(rule\ rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv[OF\text{-}inv])$
using $bj\ cdcl_W\text{-}bj.skip\ local.skip\ other\ rtrancp\text{-}mono[of\ skip\ cdcl_W]$ **by** *blast*
then have $M_T \models_{as} CNot\ ?D$
unfolding $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}conflicting\text{-}def$ **using** *T* **by** *blast*
have $\forall L \in \#?D. atm\text{-}of\ L \in atm\text{-}of\ 'lits\text{-}of\ M_T$
proof –
have $f1: \bigwedge l. \neg M_T \models_a \{\# - l\# \} \vee atm\text{-}of\ l \in atm\text{-}of\ 'lits\text{-}of\ M_T$
by $(simp\ add: atm\text{-}of\text{-}in\text{-}atm\text{-}of\text{-}set\text{-}iff\text{-}in\text{-}set\text{-}or\text{-}uminus\text{-}in\text{-}set\ in\text{-}lit\text{-}of\text{-}true\text{-}annot$
 $lits\text{-}of\text{-}def)$
have $\bigwedge l. l \notin \# D \vee - l \in lits\text{-}of\ M_T$
using $\langle M_T \models_{as} CNot (D + \{\#L\#\}) \rangle\ multi\text{-}member\text{-}split$ **by** *fastforce*
then show *?thesis*
using *f1* **by** $(meson\ \langle M_T \models_{as} CNot (D + \{\#L\#\}) \rangle\ ball\text{-}msetI\ true\text{-}annots\text{-}CNot\text{-}all\text{-}atms\text{-}defined)$
qed

moreover have *no-dup M*
using *inv S* **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *auto*
ultimately have $\forall L \in \#?D. atm\text{-}of\ L \notin atm\text{-}of\ 'lits\text{-}of\ MS$
unfolding *M* **unfolding** *lits-of-def* **by** *auto*
then have $H: \bigwedge L. L \in \#?D \implies get\text{-}level\ L\ M = get\text{-}level\ L\ M_T$
unfolding *M* **by** $(fastforce\ simp: lits\text{-}of\text{-}def)$
have $[simp]: get\text{-}maximum\text{-}level\ ?D\ M = get\text{-}maximum\text{-}level\ ?D\ M_T$
by $(metis\ \langle M_T \models_{as} CNot (D + \{\#L\#\}) \rangle\ M\ nm\ ball\text{-}msetI\ true\text{-}annots\text{-}CNot\text{-}all\text{-}atms\text{-}defined$
 $get\text{-}maximum\text{-}level\text{-}skip\text{-}un\text{-}marked\text{-}not\text{-}present)$

have *lev-l'*: $get\text{-}level\ L\ M_T = k$
using *lev-l* **by** $(auto\ simp: H)$
have $[simp]: trail (reduce\text{-}trail\text{-}to\ M1\ T) = M1$

```

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 cdclW-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) cdclW-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: $\text{trail } S = M_0 @ \text{trail } U$ **and**
nm: $\forall m \in \text{set } M_0. \neg \text{is-marked } m$
using *rtrancp-skip-state-decomp*[*OF skip*] **by** *blast*

obtain $M' D' L' i K M1 M2$ **where**
S': $\text{state } S = (M', N, U', k, C\text{-Clause } (D' + \{\#L'\#\}))$ **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: $\text{state } T = (\text{Propagated } L' (D' + \{\#L'\#\}) \# M1, N, \{\#D' + \{\#L'\#\}\# + U', i, C\text{-True})$
using *bt* $\langle \text{cdcl}_W\text{-M-level-inv } S \rangle S$ **by** $(\text{elim backtrack-levE}) \text{ fastforce+}$
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_W-all-struct-inv-def* *cdcl_W-M-level-inv-def* **by** *auto*
then have $i < k$
unfolding M **by** $(\text{force simp add: rev-swap[symmetric] dest!: arg-cong[of - - set]})$

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** $(\text{metis add-diff-cancel-right' diff-single-trivial diff-union-swap multi-self-add-other-not-self})$
moreover
have $M': M' = M_0 @ \text{Propagated } L ((C + \{\#L\#\})) \# M$
using *tr-S* $U S S'$ **by** $(\text{auto simp: lits-of-def})$
have *no-dup* M'
using *inv* $U S'$ **unfolding** *cdcl_W-all-struct-inv-def* *cdcl_W-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** $(\text{auto simp: lits-of-def})$
have *get-all-levels-of-marked* $M' = \text{rev } [1..<1+k]$
using *inv* $U S'$ **unfolding** *cdcl_W-all-struct-inv-def* *cdcl_W-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** $(\text{simp add: get-all-levels-of-marked-no-marked})$
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** $(\text{auto simp: lits-of-def})$
then have *get-level* $L M' = k$
using *get-rev-level-notin-end*[*of* $L \text{ rev } M_0 0$
 $\text{rev } M @ \text{Propagated } L ((C + \{\#L\#\})) \# []$]
using *tr-S* *get-lev-L* $M' U S'$ **by** $(\text{simp add: nm lits-of-def})$
ultimately have *get-maximum-level* $D' M' \geq k$
by $(\text{metis get-maximum-level-ge-get-level get-rev-level-uminus})$
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 cdclW** S U
  using bj cdclW-bj.skip local.skip mono-rtrancp[of skip cdclW S U] other by meson
then have cdclW-all-struct-inv U
  using inv rtrancp-cdclW-all-struct-inv-inv by blast
then have Propagated L ( (C + {#L#})) # M  $\models$ as CNot (D' + {#L'#})
  using cdclW-all-struct-inv-def cdclW-conflicting-def U by auto
then have  $\forall L' \in \#D. \text{atm-of } L' \in \text{atm-of ' lits-of (Propagated L ( (C + {#L\#})) \# M)$ 
  by (metis CNot-plus CNot-singleton Un-insert-right  $\langle D = D' \rangle$  true-annots-insert ball-msetI
    atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-CNot-implies-uminus(2)
    sup-bot.comm-neutral)
then have get-maximum-level D M' = k
  using tr-S nm U S'
    get-maximum-level-skip-un-marked-not-present[of D
      Propagated L ( (C + {#L#})) \# M M0]
  unfolding  $\langle \text{get-maximum-level } D \text{ (Propagated L ( (C + \{ \#L\# \}) \# M) = k)$ 
  unfolding  $\langle D = D' \rangle$ 
  by simp
show False
  using  $\langle \text{get-maximum-level } D' M' = i \rangle \langle \text{get-maximum-level } D M' = k \rangle \langle i < k \rangle$  by auto
qed

```

lemma *if-can-apply-resolve-no-more-backtrack:*

```

assumes
  skip: skip** S U and
  resolve: resolve S T and
  inv: cdclW-all-struct-inv S
shows  $\neg \text{backtrack } U V$ 
using assms
by (meson if-can-apply-backtrack-no-more-resolve rtrancp.rtrancp-refl
  rtrancp-skip-backtrack-backtrack)

```

lemma *if-can-apply-backtrack-skip-or-resolve-is-skip:*

```

assumes
  bt: backtrack S T and
  skip: skip-or-resolve** S U and
  inv: cdclW-all-struct-inv S
shows skip** S U
using assms(2,3,1)
by induction (simp-all add: if-can-apply-backtrack-no-more-resolve)

```

lemma *cdcl_W-bj-bj-decomp:*

```

assumes cdclW-bj** S W and cdclW-all-struct-inv S
shows
  ( $\exists T U V. (\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S)^{**} S T$ 
     $\wedge (\lambda T U. \text{resolve } T U \wedge \text{no-step backtrack } T) T U$ 
     $\wedge \text{skip}^{**} U V \wedge \text{backtrack } V W$ )
 $\vee (\exists T U. (\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S)^{**} S T$ 
   $\wedge (\lambda T U. \text{resolve } T U \wedge \text{no-step backtrack } T) T U \wedge \text{skip}^{**} U W)$ 
 $\vee (\exists T. \text{skip}^{**} S T \wedge \text{backtrack } T W)$ 
 $\vee \text{skip}^{**} S W$  (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-rtrancp[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
    rtrancp-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'⟩ rtrancp-cdclW-all-struct-inv-inv
      cdclW-o.bj rtrancp-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-rtrancp 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-rtrancp-into-rtrancp)
  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

```

The case distinction is needed, since $T \sim V$ does not imply that $R^{**} T V$.

lemma *cdcl_W-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 ~ V ∨ cdclW-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)  $cdcl_W\text{-bj}^{**} T V$ 
using IH by blast
then show ?case
proof cases
  case TV
  have no-step  $cdcl_W\text{-bj } T$ 
    using  $\langle cdcl_W\text{-M-level-inv } T \rangle$  n-s  $cdcl_W\text{-bj-state-eq-compatible[of } T - V]$  TV by auto
  then show ?thesis
    using s-o-r by auto
next
  case bj-TV
  then obtain  $U'$  where
     $T\text{-}U'$ :  $cdcl_W\text{-bj } T U'$  and
     $cdcl_W\text{-bj}^{**} U' V$ 
    using IH n-s s-o-r by (metis rtranclp-unfold tranclpD)
  have  $cdcl_W^{**} S T$ 
    by (metis (no-types, hide-lams) bj mono-rtranclp[of  $cdcl_W\text{-bj } cdcl_W$ ] other st)
  then have  $inv\text{-}T$ :  $cdcl_W\text{-all-struct-inv } T$ 
    by (metis (no-types, hide-lams) inv rtranclp- $cdcl_W\text{-all-struct-inv-inv}$ )

  have  $lev\text{-}U$ :  $cdcl_W\text{-M-level-inv } U$ 
    using s-o-r  $cdcl_W\text{-consistent-inv } lev\text{-}T$  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 -  $inv\text{-}T$ ]
       $cdcl_W\text{-bj-decomp-resolve-skip-and-bj}$ 
      by (meson bj-TV  $cdcl_W\text{-bj.backtrack } inv\text{-}T lev\text{-}T$  n-s
         $rtranclp\text{-skip-backtrack-backtrack-end}$ )
    then have  $cdcl_W\text{-bj}^{**} T V0$  and  $cdcl_W\text{-bj } V0 V$ 
      using rtranclp-mono[of  $skip cdcl_W\text{-bj}$ ] by blast+
    then show ?thesis
      using  $\langle backtrack V0 V \rangle \langle skip^{**} T V0 \rangle$  backtrack-unique  $inv\text{-}T$  local.backtrack
       $rtranclp\text{-skip-backtrack-backtrack}$  by auto
    next
    case resolve
    then have  $U \sim U'$ 
      by (meson  $T\text{-}U'$   $cdcl_W\text{-bj.simps}$  if-can-apply-backtrack-no-more-resolve  $inv\text{-}T$ 
        resolve-skip-deterministic resolve-unique rtranclp.rtrancl-refl)
    then show ?thesis
      using  $\langle cdcl_W\text{-bj}^{**} U' V \rangle$  unfolding rtranclp-unfold
      by (meson  $T\text{-}U'$  bj  $cdcl_W\text{-consistent-inv } lev\text{-}T$  other state-eq-ref state-eq-sym
         $tranclp\text{-}cdcl_W\text{-bj-state-eq-compatible}$ )
    next
    case skip
    consider
      (sk)  $skip T U'$ 
      | (bt) backtrack  $T U'$ 
      using  $T\text{-}U'$  by (meson  $cdcl_W\text{-bj.cases}$  local.skip resolve-skip-deterministic)
    then show ?thesis
      proof cases

```

```

    case sk
  then show ?thesis
    using  $\langle \text{cdcl}_W\text{-bj}^{**} \ U' \ V \rangle$  unfolding rtranclp-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  $\langle \text{cdcl}_W\text{-bj}^{**} \ U' \ V \rangle$  backtrack inv-T trancpl-unfold-begin
        rtranclp-skip-backtrack-backtrack-end trancpl-into-rtranclp)
qed
qed
qed
qed

```

lemma *cdcl_W-bj-unique-normal-form*:

```

assumes
  ST: cdclW-bj** S T and SU: cdclW-bj** S U and
  n-s-U: no-step cdclW-bj U and
  n-s-T: no-step cdclW-bj T and
  inv: cdclW-all-struct-inv S
shows T ~ U

```

proof –

```

have T ~ U  $\vee$  cdclW-bj** T U
  using ST SU cdclW-bj-strongly-confluent inv n-s-U by blast
then show ?thesis
  by (metis (no-types) n-s-T rtranclp-unfold state-eq-ref trancpl-unfold-begin)

```

qed

lemma *full-cdcl_W-bj-unique-normal-form*:

```

assumes full cdclW-bj S T and full cdclW-bj S U and
  inv: cdclW-all-struct-inv S
shows T ~ U
  using cdclW-bj-unique-normal-form assms unfolding full-def by blast

```

19.4 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'* |

fw-r-conflict: *conflict S T* \Longrightarrow *full cdcl_W-bj T U* \Longrightarrow *cdcl_W-merge-restart S U* |

fw-r-decide: *decide S S'* \Longrightarrow *cdcl_W-merge-restart S S'* |

fw-r-rf: *cdcl_W-rf S S'* \Longrightarrow *cdcl_W-merge-restart S S'*

lemma *cdcl_W-merge-restart-cdcl_W*:

```

assumes cdclW-merge-restart S T
shows cdclW** S T
  using assms

```

proof *induction*

```

case (fw-r-conflict S T U) note confl = this(1) and bj = this(2)

```

```

have cdclW S T using confl by (simp add: cdclW.intros r-into-rtranclp)

```

moreover

```

have cdclW-bj** T U using bj unfolding full-def by auto

```

```

then have cdclW** T U by (metis cdclW-o.bj mono-rtranclp other)

```

```

ultimately show ?case by auto
qed (simp-all add: cdclW-o.intros cdclW.intros r-into-rtrancp)

lemma cdclW-merge-restart-conflicting-true-or-no-step:
  assumes cdclW-merge-restart S T
  shows conflicting T = C-True  $\vee$  no-step cdclW 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 cdclW U V and conflicting U = C-Clause D
    then have False
      using n-s unfolding full-def
      by (induction rule: cdclW-all-rules-induct) (auto dest!: cdclW-bj.intros )
  }
  then show ?case by (cases conflicting U) fastforce+
qed (auto simp add: cdclW-rf.simps)

inductive cdclW-merge :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  fw-propagate: propagate S S'  $\Longrightarrow$  cdclW-merge S S' |
  fw-conflict: conflict S T  $\Longrightarrow$  full cdclW-bj T U  $\Longrightarrow$  cdclW-merge S U |
  fw-decide: decide S S'  $\Longrightarrow$  cdclW-merge S S' |
  fw-forget: forget S S'  $\Longrightarrow$  cdclW-merge S S'

lemma cdclW-merge-cdclW-merge-restart:
  cdclW-merge S T  $\Longrightarrow$  cdclW-merge-restart S T
  by (meson cdclW-merge.cases cdclW-merge-restart.simps forget)

lemma rtrancp-cdclW-merge-trancp-cdclW-merge-restart:
  cdclW-merge** S T  $\Longrightarrow$  cdclW-merge-restart** S T
  using rtrancp-mono[of cdclW-merge cdclW-merge-restart] cdclW-merge-cdclW-merge-restart by blast

lemma cdclW-merge-rtrancp-cdclW:
  cdclW-merge S T  $\Longrightarrow$  cdclW** S T
  using cdclW-merge-cdclW-merge-restart cdclW-merge-restart-cdclW by blast

lemma rtrancp-cdclW-merge-rtrancp-cdclW:
  cdclW-merge** S T  $\Longrightarrow$  cdclW** S T
  using rtrancp-mono[of cdclW-merge cdclW**] cdclW-merge-rtrancp-cdclW by auto

lemma cdclW-merge-is-cdclNOT-merged-bj-learn:
  assumes
    inv: cdclW-all-struct-inv S and
    cdclW:cdclW-merge S T
  shows cdclNOT-merged-bj-learn S T
     $\vee$  (no-step cdclW-merge T  $\wedge$  conflicting T  $\neq$  C-True)
  using cdclW 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 CNot C and
    undef: undefined-lit (trail S) L and
    T: T  $\sim$  cons-trail (Propagated L (C + {#L#})) S

```



```

    using propa by auto
have propagateNOT S T
  apply (rule propagateNOT.propagateNOT[of - C L])
  using H CL T undef M-C by (auto simp: state-eqNOT-def state-eq-def clauses-def
    simp del: state-simp)
then show ?case
  using cdclNOT-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 ∈ atms-of-msu (init-clss S) and
  T: T ∼ cons-trail (Marked L (Suc (backtrack-lvl S)))
  (update-backtrack-lvl (Suc (backtrack-lvl S)) S)
  by auto
have decideNOT S T
  apply (rule decideNOT.decideNOT)
  using undef-L apply simp
  using atm-L inv unfolding cdclW-all-struct-inv-def no-strange-atm-def clauses-def apply auto[]
  using T undef-L unfolding state-eq-def state-eqNOT-def by (auto simp: clauses-def)
then show ?case using cdclNOT-merged-bj-learn-decideNOT 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
  ¬ M ⊨asm clauses S and
  C ∉ set (get-all-mark-of-propagated (trail S)) and
  C-init: C ∉# init-clss S and
  C-le: C ∈# learned-clss S and
  T: T ∼ remove-cls C S
  by auto
have init-clss S ⊨pm C
  using inv C-le unfolding cdclW-all-struct-inv-def cdclW-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 ⊨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-cls diff-zero grOI
    init-clss-remove-cls learned-clss-remove-cls plus-multiset.rep-eq replicate-mset-0
    semiring-normalization-rules(5))
have forgetNOT S T
  apply (rule forgetNOT.forgetNOT)
  using S-C apply blast
  using S apply simp
  using ⟨C ∈# learned-clss S⟩ 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)
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 CS where
  confl-T: conflicting T = C-Clause CS and
  CS: CS ∈# clauses S and

```

```
|S-CS: trail S  $\models$ as CNot CS
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 rtranclp-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-True
    using confl by (induction rule: rtranclp-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-rtranclp[of skip-or-resolve cdclW] rtranclp-skip-or-resolve-rtranclp-cdclW
    by blast
  then have cdclW-M-level-inv T'
    using rtranclp-cdclW-consistent-inv (cdclW-M-level-inv T) by blast
  then obtain M1 M2 i D L K where
    confl-T': conflicting T' = C-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  $\sim$  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 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-rtranclp rtranclp-cdclW-all-struct-inv-inv
    rtranclp-cdclW-cp-rtranclp-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-msu (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 fastforce
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 add: comp-def)
    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)[]
    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* $\langle \text{tl } (\text{trail } U) \models_{\text{as}} C \text{Not } D \rangle$ *convert-trail-from-W-true-annots*)
using *inv-T' inv-U U confl-T' undef-L M1-M2* **unfolding** *cdcl_W-all-struct-inv-def*
distinct-cdcl_W-state-def **by** (*simp add: cdcl_W-M-level-inv-decomp backjump-l-cond-def*)
then show *?thesis* **using** *cdcl_{NOT}-merged-bj-learn-backjump-l* **by** *fast*
qed
qed

abbreviation *cdcl_{NOT}-restart* **where**
cdcl_{NOT}-restart \equiv *restart-ops.cdcl_{NOT}-raw-restart cdcl_{NOT} restart*

lemma *cdcl_W-merge-restart-is-cdcl_{NOT}-merged-bj-learn-restart-no-step*:

assumes
inv: cdcl_W-all-struct-inv S and
cdcl_W:cdcl_W-merge-restart S T
shows *cdcl_{NOT}-restart** S T \vee (no-step cdcl_W-merge T \wedge conflicting T \neq C-True)*

proof –

consider

(fw) cdcl_W-merge S T
| (fw-r) restart S T
using *cdcl_W* **by** (*meson cdcl_W-merge-restart.simps cdcl_W-rf.cases fw-conflict fw-decide fw-forget*
fw-propagate)

then show *?thesis*

proof *cases*

case *fw*

then have *IH: cdcl_{NOT}-merged-bj-learn S T \vee (no-step cdcl_W-merge T \wedge conflicting T \neq C-True)*

using *inv cdcl_W-merge-is-cdcl_{NOT}-merged-bj-learn* **by** *blast*

have *invS: inv_{NOT} S*

using *inv* **unfolding** *cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def* **by** *auto*

have *ff2: cdcl_{NOT}⁺⁺ S T \longrightarrow cdcl_{NOT}^{**} S T*

by (*meson tranclp-into-rtranclp*)

have *ff3: no-dup (convert-trail-from-W (trail S))*

using *invS* **by** (*simp add: comp-def*)

have *cdcl_{NOT} \leq cdcl_{NOT}-restart*

by (*auto simp: restart-ops.cdcl_{NOT}-raw-restart.simps*)

then show *?thesis*

using *ff3 ff2 IH cdcl_{NOT}-merged-bj-learn-is-tranclp-cdcl_{NOT}*

rtranclp-mono[of cdcl_{NOT} cdcl_{NOT}-restart] invS predicate2D **by** *blast*

next

case *fw-r*

then show *?thesis* **by** (*blast intro: restart-ops.cdcl_{NOT}-raw-restart.intros*)

qed

qed

abbreviation $\mu_{FW} :: 'st \Rightarrow nat$ **where**

$\mu_{FW} S \equiv$ (*if no-step cdcl_W-merge S then 0 else 1 + μ_{CDCL} '-merged (set-mset (init-clss S)) S*)

lemma *cdcl_W-merge- μ_{FW} -decreasing*:

assumes

inv: cdcl_W-all-struct-inv S and

fw: cdcl_W-merge S T

shows $\mu_{FW} T < \mu_{FW} S$

proof –

let *?A = init-clss S*

have *atm-clauses: atms-of-msu (clauses S) \subseteq atms-of-msu ?A*

using *inv* **unfolding** *cdcl_W-all-struct-inv-def no-strange-atm-def clauses-def* **by** *auto*

```

have atm-trail: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-msu ?A
  using inv unfolding cdclW-all-struct-inv-def no-strange-atm-def clauses-def by auto
have n-d: no-dup (trail S)
  using inv unfolding cdclW-all-struct-inv-def by (auto simp: cdclW-M-level-inv-decomp)
have [simp]:  $\neg$  no-step cdclW-merge S
  using fw by auto
have [simp]: init-clss S = init-clss T
  using cdclW-merge-restart-cdclW[of S T] inv rtrancp-cdclW-init-clss
  unfolding cdclW-all-struct-inv-def
  by (meson cdclW-merge.simps cdclW-merge-restart.simps cdclW-rf.simps fw)
consider
  (merged) cdclNOT-merged-bj-learn S T
| (n-s) no-step cdclW-merge T
  using cdclW-merge-is-cdclNOT-merged-bj-learn inv fw by blast
then show ?thesis
  proof cases
    case merged
    then show ?thesis
      using cdclNOT-decreasing-measure'[OF - - atm-clauses] atm-trail n-d
      by (auto split: split-if simp: comp-def)
    next
    case n-s
    then show ?thesis by simp
  qed
qed

lemma wf-cdclW-merge: wf {(T, S). cdclW-all-struct-inv S  $\wedge$  cdclW-merge S T}
  apply (rule wfP-if-measure[of - -  $\mu_{FW}$ ])
  using cdclW-merge- $\mu_{FW}$ -decreasing by blast

lemma cdclW-all-struct-inv-trancp-cdclW-merge-trancp-cdclW-merge-cdclW-all-struct-inv:
  assumes
    inv: cdclW-all-struct-inv b
    cdclW-merge++ b a
  shows ( $\lambda S T$ . cdclW-all-struct-inv S  $\wedge$  cdclW-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 cdclW-all-struct-inv c
    using trancp-into-rtrancp[OF st] cdclW-merge-rtrancp-cdclW
    assms(1) rtrancp-cdclW-all-struct-inv-inv rtrancp-mono[of cdclW-merge cdclW**] by fastforce
  then have ( $\lambda S T$ . cdclW-all-struct-inv S  $\wedge$  cdclW-merge S T)++ c d
    using fw by auto
  then show ?case using IH by auto
qed

lemma wf-trancp-cdclW-merge: wf {(T, S). cdclW-all-struct-inv S  $\wedge$  cdclW-merge++ S T}
  using wf-trancp[OF wf-cdclW-merge]
  apply (rule wf-subset)
  by (auto simp: trancp-set-trancp
    cdclW-all-struct-inv-trancp-cdclW-merge-trancp-cdclW-merge-cdclW-all-struct-inv)

```

lemma *backtrack-is-full1-cdcl_W-bj*:
 assumes *bt*: *backtrack S T* and *inv*: *cdcl_W-M-level-inv S*
 shows *full1 cdcl_W-bj S T*
proof –
 have *no-step cdcl_W-bj T*
 using *bt inv backtrack-no-cdcl_W-bj* by *blast*
 moreover have *cdcl_W-bj⁺⁺ S T*
 using *bt* by *auto*
 ultimately show *?thesis unfolding full1-def* by *blast*
qed

lemma *rtranc1-cdcl_W-conflicting-true-cdcl_W-merge-restart*:
 assumes *cdcl_W^{**} S V* and *inv*: *cdcl_W-M-level-inv S* and *conflicting S = C-True*
 shows *(cdcl_W-merge-restart^{**} S V ∧ conflicting V = C-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_W = this(2)* and *IH = this(3)[OF this(4–)]* and
confl[simp] = this(5) and *inv = this(4)*
 from *cdcl_W*
 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_W-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 cdcl_W-merge-restart.fw-r-decide[of U V]* by *auto*
next
 case *bj*
 moreover {
 assume *skip-or-resolve U V*
 have *f1: cdcl_W-bj⁺⁺ U V*
 by *(simp add: local.bj tranc1p.r-into-tranc1)*
 obtain *T T' :: 'st* where
*f2: cdcl_W-merge-restart^{**} S U*
 $\vee \text{cdcl}_W\text{-merge-restart}^{**} S T \wedge \text{conflicting } U \neq C\text{-True}$

```

     $\wedge \text{conflict } T \ T' \wedge \text{cdcl}_W\text{-bj}^{**} \ T' \ U$ 
  using IH confl by blast
then have ?thesis
proof -
  have conflicting  $V \neq C\text{-True} \wedge \text{conflicting } U \neq C\text{-True}$ 
    using  $\langle \text{skip-or-resolve } U \ V \rangle$  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  $U \neq C\text{-True}$  by auto
  then obtain  $T \ T'$  where
    cdclW-merge-restart**  $S \ T$  and
    conflicting  $U \neq C\text{-True}$  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  $V = C\text{-True}$ 
    using  $\langle \text{backtrack } U \ V \rangle$  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  $\langle \text{cdcl}_W\text{-bj}^{**} \ T' \ U \rangle$  apply fast
    using  $\langle \text{backtrack } U \ V \rangle$  backtrack-is-full1-cdclW-bj invU unfolding full1-def full-def
    by blast
  then have ?thesis
    using cdclW-merge-restart.fw-r-conflict[of  $T \ T' \ V$ ]  $\langle \text{conflict } T \ T' \rangle$ 
     $\langle \text{cdcl}_W\text{-merge-restart}^{**} \ S \ T \rangle$   $\langle \text{conflicting } V = C\text{-True} \rangle$  by auto
}
ultimately show ?thesis by (auto simp: cdclW-bj.simps)
qed
next
case rf
moreover then have conflicting  $U = C\text{-True}$  and conflicting  $V = C\text{-True}$ 
  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-cdclW-no-step-cdclW-merge-restart: no-step cdclW  $S \implies \text{no-step } \text{cdcl}_W\text{-merge-restart } S$ 
  by (auto simp: cdclW.simps cdclW-merge-restart.simps cdclW-o.simps cdclW-bj.simps)

lemma no-step-cdclW-merge-restart-no-step-cdclW:
  assumes
    conflicting  $S = C\text{-True}$  and
    cdclW-M-level-inv  $S$  and
    no-step cdclW-merge-restart  $S$ 
  shows no-step cdclW  $S$ 
proof -
  { fix  $S'$ 

```

```

    assume conflict S S'
    then have cdclW S S' using cdclW.conflict by auto
    then have cdclW-M-level-inv S'
      using assms(2) cdclW-consistent-inv by blast
    then obtain S'' where full cdclW-bj S' S''
      using cdclW-bj-exists-normal-form[of S'] by auto
    then have False
      using ⟨conflict S S'⟩ assms(3) fw-r-conflict by blast
  }
  then show ?thesis
    using assms unfolding cdclW.simps cdclW-merge-restart.simps cdclW-o.simps cdclW-bj.simps
    by fastforce
qed

```

```

lemma rtrancpl-cdclW-merge-restart-no-step-cdclW-bj:
  assumes
    cdclW-merge-restart** S T and
    conflicting S = C-True
  shows no-step cdclW-bj T
  using assms
  apply (induction rule: rtrancpl-induct)
  apply (fastforce simp: cdclW-bj.simps cdclW-rf.simps cdclW-merge-restart.simps full-def)
  apply (fastforce simp: cdclW-bj.simps cdclW-rf.simps cdclW-merge-restart.simps full-def)
  done

```

If $\text{conflicting } S \neq C\text{-True}$, 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-true-full-cdclW-iff-full-cdclW-merge:
  assumes confl: conflicting S = C-True and lev: cdclW-M-level-inv S
  shows full cdclW S V  $\longleftrightarrow$  full cdclW-merge-restart S V
proof
  assume full: full cdclW-merge-restart S V
  then have st: cdclW** S V
    using rtrancpl-mono[of cdclW-merge-restart cdclW**] cdclW-merge-restart-cdclW
    unfolding full-def by auto

  have n-s: no-step cdclW-merge-restart V
    using full unfolding full-def by auto
  have n-s-bj: no-step cdclW-bj V
    using rtrancpl-cdclW-merge-restart-no-step-cdclW-bj confl full unfolding full-def by auto
  have  $\bigwedge S'. \text{conflict } V S' \implies \text{cdcl}_W\text{-M-level-inv } S'$ 
    using cdclW.conflict cdclW-consistent-inv lev rtrancpl-cdclW-consistent-inv st by blast
  then have  $\bigwedge S'. \text{conflict } V S' \implies \text{False}$ 
    using n-s n-s-bj cdclW-bj-exists-normal-form cdclW-merge-restart.simps by meson
  then have n-s-cdclW: no-step cdclW 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)  $cdcl_W$ -merge-restart**  $S$   $V$  and conflicting  $V = C$ -True
| (bj)  $T$   $U$  where
   $cdcl_W$ -merge-restart**  $S$   $T$  and
  conflicting  $V \neq C$ -True and
  conflict  $T$   $U$  and
   $cdcl_W$ -bj**  $U$   $V$ 
using full  $rtrancl$ - $cdcl_W$ -conflicting-true- $cdcl_W$ -merge-restart  $confl$   $lev$  unfolding full-def
by meson
then have  $cdcl_W$ -merge-restart**  $S$   $V$ 
proof cases
  case fw
    then show ?thesis by fast
  next
    case (bj  $T$   $U$ )
    have no-step  $cdcl_W$ -bj  $V$ 
      using full unfolding full-def by (meson  $cdcl_W$ -o.bj other)
    then have full  $cdcl_W$ -bj  $U$   $V$ 
      using  $\langle cdcl_W$ -bj**  $U$   $V \rangle$  unfolding full-def by auto
    then have  $cdcl_W$ -merge-restart  $T$   $V$ 
      using  $\langle conflict$   $T$   $U \rangle$   $cdcl_W$ -merge-restart.fw-r-conflict by blast
    then show ?thesis using  $\langle cdcl_W$ -merge-restart**  $S$   $T \rangle$  by auto
  qed
ultimately show full  $cdcl_W$ -merge-restart  $S$   $V$  unfolding full-def by fast
qed

```

lemma *init-state-true-full- $cdcl_W$ -iff-full- $cdcl_W$ -merge:*
shows full $cdcl_W$ (init-state N) $V \longleftrightarrow$ full $cdcl_W$ -merge-restart (init-state N) V
by (rule conflicting-true-full- $cdcl_W$ -iff-full- $cdcl_W$ -merge) auto

19.5 FW with strategy

19.5.1 The intermediate step

inductive $cdcl_W$ -s' :: 'st \Rightarrow 'st \Rightarrow bool **where**
conflict': full1 $cdcl_W$ -cp S $S' \Longrightarrow cdcl_W$ -s' S S' |
decide': decide S $S' \Longrightarrow$ no-step $cdcl_W$ -cp $S \Longrightarrow$ full $cdcl_W$ -cp S' $S'' \Longrightarrow cdcl_W$ -s' S S'' |
bj': full1 $cdcl_W$ -bj S $S' \Longrightarrow$ no-step $cdcl_W$ -cp $S \Longrightarrow$ full $cdcl_W$ -cp S' $S'' \Longrightarrow cdcl_W$ -s' S S''

inductive-cases $cdcl_W$ -s'E: $cdcl_W$ -s' S T

lemma *rtranclp- $cdcl_W$ -bj-full1- $cdclp$ - $cdcl_W$ -stgy:*
 $cdcl_W$ -bj** S $S' \Longrightarrow$ full $cdcl_W$ -cp S' $S'' \Longrightarrow cdcl_W$ -stgy** S S''
proof (induction rule: converse-rtranclp-induct)
case base
then show ?case **by** (metis $cdcl_W$ -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 $cdcl_W$ -cp T
using bj **by** (auto simp add: $cdcl_W$ -bj.simps)
consider
 (U) $U = S'$
 | (U') U' **where** $cdcl_W$ -bj U U' **and** $cdcl_W$ -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-rtrancp)
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-rtrancp-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-rtrancp)
  by (metis full1-def rtrancp-cdclW-bj-full1-cdclp-cdclW-stgy trancp-into-rtrancp)

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: rtrancp-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-rtrancp[of cdclW-bj cdclW T T''] other
  st rtrancp.rtrancp-into-rtrancp)
then have inv-T'': cdclW-all-struct-inv T''
  using inv rtrancp-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 cdclW-bj T Z
      by (meson trancpD ⟨cdclW-bj++ T T'⟩)
    { assume cdclW-cp++ T U
      then obtain Z' where cdclW-cp T Z'
        by (meson trancpD)
      then have False
        using ⟨cdclW-bj T Z⟩ by (fastforce simp: cdclW-bj.simps cdclW-cp.simps)
    }
  end

```

```

    then show ?thesis
      using full unfolding full-def rtrancpl-unfold by blast
  qed
obtain U'' where full cdclW-cp T'' U''
  using cdclW-cp-normalized-element-all-inv inv-T'' by blast
moreover then have cdclW-stgy** U U''
  by (metis ⟨T = U⟩ ⟨cdclW-bj++ T T'⟩ rtrancpl-cdclW-bj-full1-cdclp-cdclW-stgy rtrancpl-unfold)
moreover have cdclW-s!* U U''
proof -
  obtain ss :: 'st ⇒ 'st where
    f1: ∀ x2. (∃ v3. cdclW-cp x2 v3) = cdclW-cp x2 (ss x2)
  by maura
  have ¬ cdclW-cp U (ss U)
  by (meson full full-def)
  then show ?thesis
    using f1 by (metis (no-types) ⟨T = U⟩ ⟨full1 cdclW-bj T T'⟩ bj' calculation(1)
      r-into-rtrancpl)
  qed
ultimately show ?case
  using ⟨full1 cdclW-bj T T'⟩ ⟨full cdclW-cp T'' U''⟩ unfolding ⟨T = U⟩ by blast
qed

```

lemma *cdcl_W-cp-cdcl_W-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
  ∨ (∃ U'. full1 cdclW-bj U U' ∧ (∀ U''. full cdclW-cp U' U'' ⟶ full cdclW-cp T' U''
    ∧ cdclW-s!* U U''))
using assms(2,1,3,4)
proof (induction rule: rtrancpl-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-rtrancpl[of cdclW-bj cdclW T T''] other st
      rtrancpl.rtrancpl-into-rtrancpl)
  then have inv-T'': cdclW-all-struct-inv T''
    using inv rtrancpl-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.premis(3))
  then have T = U
  proof -
    obtain Z where cdclW-bj T Z
    by (meson trancplD ⟨cdclW-bj++ T T'⟩)
    { assume cdclW-cp++ T U
      then obtain Z' where cdclW-cp T Z'
      by (meson trancplD)
      then have False
    }
  qed

```

```

    using  $\langle \text{cdcl}_W\text{-bj } T \ Z \rangle$  by (fastforce simp:  $\text{cdcl}_W\text{-bj.simps } \text{cdcl}_W\text{-cp.simps}$ )
  }
  then show ?thesis
    using full unfolding full-def rtrancpl-unfold by blast
qed
{ fix  $U''$ 
  assume full  $\text{cdcl}_W\text{-cp } T'' \ U''$ 
  moreover then have  $\text{cdcl}_W\text{-stgy}^{**} \ U \ U''$ 
    by (metis  $\langle T = U \rangle \langle \text{cdcl}_W\text{-bj}^{++} \ T \ T'' \rangle$  rtrancpl-cdcl $_W$ -bj-full1-cdcl $_W$ -stgy rtrancpl-unfold)
  moreover have  $\text{cdcl}_W\text{-s}^{**} \ U \ U''$ 
  proof -
    obtain  $ss :: 'st \Rightarrow 'st$  where
       $f1: \forall x2. (\exists v3. \text{cdcl}_W\text{-cp } x2 \ v3) = \text{cdcl}_W\text{-cp } x2 \ (ss \ x2)$ 
    by moura
    have  $\neg \text{cdcl}_W\text{-cp } U \ (ss \ U)$ 
    by (meson assms(1) full-def)
    then show ?thesis
      using f1 by (metis (no-types)  $\langle T = U \rangle \langle \text{full1 } \text{cdcl}_W\text{-bj } T \ T'' \rangle$  bj' calculation(1)
        r-into-rtrancpl)
  qed
  ultimately have full1  $\text{cdcl}_W\text{-bj } U \ T''$  and  $\text{cdcl}_W\text{-s}^{**} \ T'' \ U''$ 
  using  $\langle \text{full1 } \text{cdcl}_W\text{-bj } T \ T'' \rangle \langle \text{full } \text{cdcl}_W\text{-cp } T'' \ U'' \rangle$  unfolding  $\langle T = U \rangle$ 
    apply blast
  by (metis  $\langle \text{full } \text{cdcl}_W\text{-cp } T'' \ U'' \rangle \text{cdcl}_W\text{-s'.simps full-unfold rtrancpl.simps}$ )
}
then show ?case
  using  $\langle \text{full1 } \text{cdcl}_W\text{-bj } T \ T'' \rangle$  full bj' unfolding  $\langle T = U \rangle$  full-def by (metis r-into-rtrancpl)
qed

```

lemma $\text{cdcl}_W\text{-stgy-cdcl}_W\text{-s'-connected}$:

```

  assumes  $\text{cdcl}_W\text{-stgy } S \ U$  and  $\text{cdcl}_W\text{-all-struct-inv } S$ 
  shows  $\text{cdcl}_W\text{-s}' \ S \ U$ 
     $\vee (\exists U'. \text{full1 } \text{cdcl}_W\text{-bj } U \ U' \wedge (\forall U''. \text{full } \text{cdcl}_W\text{-cp } U' \ U'' \longrightarrow \text{cdcl}_W\text{-s}' \ S \ U''))$ 
  using assms
proof (induction rule:  $\text{cdcl}_W\text{-stgy.induct}$ )
  case (conflict'  $T$ )
  then have  $\text{cdcl}_W\text{-s}' \ S \ T$ 
    using  $\text{cdcl}_W\text{-s'}.conflict'$  by blast
  then show ?case
    by blast
next
  case (other'  $T \ U$ ) note  $o = \text{this}(1)$  and  $n\text{-s} = \text{this}(2)$  and  $\text{full} = \text{this}(3)$  and  $\text{inv} = \text{this}(4)$ 
  show ?case
    using o
  proof cases
    case decide
    then show ?thesis using  $\text{cdcl}_W\text{-s'.simps full } n\text{-s}$  by blast
  next
    case bj
    have  $\text{inv-T}: \text{cdcl}_W\text{-all-struct-inv } T$ 
    using  $\text{cdcl}_W\text{-all-struct-inv-inv } o$  other other'.prems by blast
    consider
      (cp) full  $\text{cdcl}_W\text{-cp } T \ U$  and no-step  $\text{cdcl}_W\text{-bj } T$ 
      | (fbj)  $T'$  where full1  $\text{cdcl}_W\text{-bj } T \ T'$ 
    apply (cases no-step  $\text{cdcl}_W\text{-bj } T$ )

```

```

    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
  end
end

```

```

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_W-stgy-cdcl_W-s'-no-step:*
assumes *cdcl_W-stgy S U and cdcl_W-all-struct-inv S and no-step cdcl_W-bj U*
shows *cdcl_W-s' S U*
using *cdcl_W-stgy-cdcl_W-s'-connected[OF assms(1,2)] assms(3)*
by (metis (no-types, lifting) full1-def tranclpD)

lemma *rtranclp-cdcl_W-stgy-connected-to-rtranclp-cdcl_W-s':*
assumes *cdcl_W-stgy** S U and inv: cdcl_W-M-level-inv S*
shows *cdcl_W-s** S U ∨ (∃ T. cdcl_W-s** S T ∧ cdcl_W-bj⁺⁺ T U ∧ conflicting U ≠ 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_W-s' T V*

using *cdcl_W-s'.conflict'* **by** *blast*

obtain *ss :: 'st* **where**

*f3: S = T ∨ cdcl_W-stgy** S ss ∧ cdcl_W-stgy ss T*

by (metis (full-types) rtranclp.simps st)

obtain *ssa :: 'st* **where**

cdcl_W-cp T ssa

using *conflict'* **by** (metis (no-types) full1-def tranclpD)

then have *S = T*

using *f3* **by** (metis (no-types) cdcl_W-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') 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'
    have cdclW-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  $S\text{-}S' = \text{this}(1)$  and  $\text{bj-}T = \text{this}(2)$ 
have  $\text{cdcl}_W\text{-bj}^{++} S' V$ 
  using skip  $\text{bj-}T$  by (metis  $\langle U = V \rangle \text{cdcl}_W\text{-bj.skip tranclp.simps}$ )

  moreover have conflicting  $V \neq C\text{-True}$ 
    using skip by auto
  ultimately show ?thesis using  $S\text{-}S'$  by auto
qed
next
case resolve
then have [simp]:  $U = V$ 
  using full converse-rtranclpE unfolding full-def by fastforce
consider
  ( $s'$ )  $\text{cdcl}_W\text{-s}^{**} S T$ 
| ( $\text{bj}$ )  $S'$  where  $\text{cdcl}_W\text{-s}^{**} S S'$  and  $\text{cdcl}_W\text{-bj}^{++} S' T$  and conflicting  $T \neq C\text{-True}$ 
  using IH by blast
then show ?thesis
proof cases
case  $s'$ 
have  $\text{cdcl}_W\text{-bj}^{++} T V$ 
  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  $S\text{-}S' = \text{this}(1)$  and  $\text{bj-}T = \text{this}(2)$ 
have  $\text{cdcl}_W\text{-bj}^{++} S' V$ 
  using resolve  $\text{bj-}T$  by (metis  $\langle U = V \rangle \text{cdcl}_W\text{-bj.resolve tranclp.simps}$ )
moreover have conflicting  $V \neq C\text{-True}$ 
  using resolve by auto
ultimately show ?thesis using  $S\text{-}S'$  by auto
qed
qed
qed
qed

lemma  $n\text{-step-cdcl}_W\text{-stgy-iff-no-step-cdcl}_W\text{-cl-cdcl}_W\text{-o}$ :
  assumes  $\text{inv: cdcl}_W\text{-all-struct-inv } S$ 
  shows  $\text{no-step cdcl}_W\text{-s}' S \longleftrightarrow \text{no-step cdcl}_W\text{-cp } S \wedge \text{no-step cdcl}_W\text{-o } S$  (is  $?S' S \longleftrightarrow ?C S \wedge ?O S$ )
proof
  assume  $?C S \wedge ?O S$ 
  then show  $?S' S$ 
    by (auto simp:  $\text{cdcl}_W\text{-s}'.\text{simps full1-def tranclp-unfold-begin}$ )
next
  assume  $n\text{-s: } ?S' S$ 
  have  $?C S$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then obtain  $S'$  where  $\text{cdcl}_W\text{-cp } S S'$ 
      by auto

```



```

then obtain  $T$  where  $full1\ cdcl_W\text{-}cp\ S\ T$ 
  using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv\ inv$  by (metis (no-types, lifting) full-unfold)
then show  $False$  using  $n\text{-}s\ cdcl_W\text{-}s'.conflict'$  by blast
qed
moreover have  $?O\ S$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $S'$  where  $cdcl_W\text{-}o\ S\ S'$ 
  by auto
  then obtain  $T$  where  $full1\ cdcl_W\text{-}cp\ S'\ T$ 
  using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv\ inv$ 
  by (meson  $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ n\text{-}s\ cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}s'\text{-}connected'\ cdcl_W\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step$ )
  then show  $False$  using  $n\text{-}s$  by (meson  $\langle cdcl_W\text{-}o\ S\ S' \rangle\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}s'\text{-}connected'\ cdcl_W\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step\ inv$ )
qed
ultimately show  $?C\ S \wedge ?O\ S$  by auto
qed

```

```

lemma  $cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W$ :
 $cdcl_W\text{-}s'\ S\ S' \implies cdcl_W^{++}\ S\ S'$ 
proof (induct rule:  $cdcl_W\text{-}s'.induct$ )
  case  $conflict'$ 
  then show  $?case$ 
  by (simp add:  $full1\text{-}def\ trancp\text{-}cdcl_W\text{-}cp\text{-}trancp\text{-}cdcl_W$ )
next
  case  $decide'$ 
  then show  $?case$ 
  using  $cdcl_W\text{-}stgy.simps\ cdcl_W\text{-}stgy\text{-}trancp\text{-}cdcl_W$  by (meson  $cdcl_W\text{-}o.simps$ )
next
  case ( $bj'\ Sa\ S'a\ S''$ ) note  $a2 = this(1)$  and  $a1 = this(2)$  and  $n\text{-}s = this(3)$ 
  obtain  $ss :: 'st \Rightarrow 'st \Rightarrow ('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st$  where
 $\forall x0\ x1\ x2. (\exists v3. x2\ x1\ v3 \wedge x2^{**}\ v3\ x0) = (x2\ x1\ (ss\ x0\ x1\ x2) \wedge x2^{**}\ (ss\ x0\ x1\ x2)\ x0)$ 
  by moura
  then have  $f3: \forall p\ s\ sa. \neg p^{++}\ s\ sa \vee p\ s\ (ss\ sa\ s\ p) \wedge p^{**}\ (ss\ sa\ s\ p)\ sa$ 
  by (metis (full-types)  $trancpD$ )
  have  $cdcl_W\text{-}bj^{++}\ Sa\ S'a \wedge no\text{-}step\ cdcl_W\text{-}bj\ S'a$ 
  using  $a2$  by (simp add:  $full1\text{-}def$ )
  then have  $cdcl_W\text{-}bj\ Sa\ (ss\ S'a\ Sa\ cdcl_W\text{-}bj) \wedge cdcl_W\text{-}bj^{**}\ (ss\ S'a\ Sa\ cdcl_W\text{-}bj)\ S'a$ 
  using  $f3$  by auto
  then show  $cdcl_W^{++}\ Sa\ S''$ 
  using  $a1\ n\text{-}s$  by (meson  $bj\ other\ rtrancp\text{-}cdcl_W\text{-}bj\ full1\text{-}cdclp\text{-}cdcl_W\text{-}stgy\ rtrancp\text{-}cdcl_W\text{-}stgy\text{-}rtrancp\text{-}cdcl_W\ rtrancp\text{-}into\text{-}trancp2$ )
qed

```

```

lemma  $trancp\text{-}cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W$ :
 $cdcl_W\text{-}s'^{++}\ S\ S' \implies cdcl_W^{++}\ S\ S'$ 
apply (induct rule:  $trancp.induct$ )
using  $cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W$  apply blast
by (meson  $cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W\ trancp\text{-}trans$ )

```

```

lemma  $rtrancp\text{-}cdcl_W\text{-}s'\text{-}rtrancp\text{-}cdcl_W$ :
 $cdcl_W\text{-}s'^{**}\ S\ S' \implies cdcl_W^{**}\ S\ S'$ 
using  $rtrancp\text{-}unfold[of\ cdcl_W\text{-}s'\ S\ S']\ trancp\text{-}cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W[of\ S\ S']$  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 rtrancpl-cdclW-s'-rtrancpl-cdclW[of S T] unfolding full-def by blast
  then have inv': cdclW-all-struct-inv T
    using rtrancpl-cdclW-all-struct-inv-inv inv by blast
  have cdclW-stgy** S T
    using ⟨?S'⟩ unfolding full-def
    using cdclW-s'-is-rtrancpl-cdclW-stgy rtrancpl-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 rtrancpl-cdclW-all-struct-inv-inv rtrancpl-cdclW-stgy-rtrancpl-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 rtrancpl-cdclW-stgy-connected-to-rtrancpl-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'.sims
        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!: trancplD simp: cdclW-cp.sims cdclW-bj.sims)
    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-cdclW-stgy-step:
  assumes
    conflict S T
    cdclW-all-struct-inv S
  shows  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
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-cdclW-stgy-step:
  assumes
    decide S T
    cdclW-all-struct-inv S
  shows  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
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 rtranclp-cdclW-cp-conflicting-C-Clause:
  cdclW-cp** S T  $\implies$  conflicting S = C-Clause D  $\implies$  S = T
  using rtranclpD tranclpD by fastforce

inductive cdclW-merge-cp :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  conflict'[intro]: conflict S T  $\implies$  full cdclW-bj T U  $\implies$  cdclW-merge-cp S U |
  propagate'[intro]: propagate++ S S'  $\implies$  cdclW-merge-cp S S'

lemma cdclW-merge-restart-cases[consumes 1, case-names conflict propagate]:
  assumes
    cdclW-merge-cp S U and
     $\bigwedge T. \text{conflict } S \ T \implies \text{full } \text{cdcl}_W\text{-bj } T \ U \implies P$  and
    propagate++ S U  $\implies$  P
  shows P
  using assms unfolding cdclW-merge-cp.simps by auto

lemma cdclW-merge-cp-tranclp-cdclW-merge:
  cdclW-merge-cp S T  $\implies$  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-cdcl_W rtranclp-trans tranclp-into-rtranclp)

lemma *full1-cdcl_W-bj-no-step-cdcl_W-bj:*

full1 cdcl_W-bj S T \implies no-step cdcl_W-cp S

by (*metis rtranclp-unfold cdcl_W-cp-conflicting-not-empty conflicting-clause.exhaust full1-def rtranclp-cdcl_W-merge-restart-no-step-cdcl_W-bj tranclpD*)

inductive *cdcl_W-s'-without-decide* **where**

conflict'-without-decide[intro]: full1 cdcl_W-cp S S' \implies cdcl_W-s'-without-decide S S' |

bj'-without-decide[intro]: full1 cdcl_W-bj S S' \implies no-step cdcl_W-cp S \implies full cdcl_W-cp S' S'' \implies cdcl_W-s'-without-decide S S''

lemma *rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W:*

*cdcl_W-s'-without-decide** S T \implies cdcl_W** S T*

apply (*induction rule: rtranclp-induct*)

apply *simp*

by (*meson cdcl_W-s'.simps cdcl_W-s'-tranclp-cdcl_W cdcl_W-s'-without-decide.simps rtranclp-tranclp-tranclp tranclp-into-rtranclp*)

lemma *rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W-s':*

*cdcl_W-s'-without-decide** S T \implies cdcl_W-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 *cdcl_W-s' y z*

using *a2* **by** (*metis (no-types) bj' cdcl_W-s'.conflict' cdcl_W-s'-without-decide.cases*)

then show *cdcl_W-s'*** S z*

using *a1* **by** (*meson r-into-rtranclp rtranclp-trans*)

qed

lemma *rtranclp-cdcl_W-merge-cp-is-rtranclp-cdcl_W-s'-without-decide:*

assumes

*cdcl_W-merge-cp** S V*

conflicting S = C-True

shows

*(cdcl_W-s'-without-decide** S V)*

$\vee (\exists T. \text{cdcl}_{\text{W}}\text{-s'-without-decide}^{**} S T \wedge \text{propagate}^{++} T V)$

$\vee (\exists T U. \text{cdcl}_{\text{W}}\text{-s'-without-decide}^{**} S T \wedge \text{full1 cdcl}_{\text{W}}\text{-bj } T U \wedge \text{propagate}^{**} U V)$

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: cdcl_W-merge-restart-cases*)

case *propagate*

then show *?thesis* **using** *IH* **by** (*meson rtranclp-tranclp-tranclp tranclp-into-rtranclp*)

next

case (*conflict U'*) **note** *confl = this(1)* **and** *bj = this(2)*

have *full1-U-U': full1 cdcl_W-cp U U'*

by (*simp add: conflict-is-full1-cdcl_W-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 rtranclp-mono[of propagate cdclW-cp] T'-U cdclW-cp.propagate' full1-U-U'
  rtranclp-full1I[of cdclW-cp T'] by (metis (full-types) predicate2D predicate2I
    tranclp-into-rtranclp)
  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 rtranclp-mono[of propagate cdclW-cp] T''-U cdclW-cp.propagate' full1-U-U'
  rtranclp-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' rtranclp.intros(2)[of - S T' U] by blast
  then show ?thesis
  by (metis full-unfold local.bj rtranclp.rtrancl-refl)
qed
qed
qed

```

lemma rtranclp-cdcl_W-s'-without-decide-is-rtranclp-cdcl_W-merge-cp:

assumes

cdcl_W-s'-without-decide** S V **and**

confl: conflicting S = C-True

shows

(cdcl_W-merge-cp** S V ∧ conflicting V = C-True)

∨ (cdcl_W-merge-cp** S V ∧ conflicting V ≠ C-True ∧ no-step cdcl_W-cp V ∧ no-step cdcl_W-bj V)

∨ (∃ T. cdcl_W-merge-cp** S T ∧ 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: cdclW-s'-without-decide.cases)
  case conflict'-without-decide
  then have rt: cdclW-cp++ U V unfolding full1-def by fast
  then have conflicting U = C-True
    using tranclp-cdclW-cp-propagate-with-conflict-or-not[of U V]
    conflict by (auto dest!: tranclpD simp: rtranclp-unfold)
  then have cdclW-merge-cp** S U using IH by auto
  consider
    (propa) propagate++ U V
  | (confl') conflict U V
  | (propa-confl') U' where propagate++ U U' conflict U' V
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[OF rt] unfolding rtranclp-unfold
  by fastforce
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 tranclp-unfold-end by auto
  ultimately show ?thesis using ⟨cdclW-merge-cp** S U⟩ by force
next
  case confl'
  then show ?thesis using ⟨cdclW-merge-cp** S U⟩ by auto
next
  case propa-confl' note propa = this(1) and confl' = this(2)
  then have cdclW-merge-cp U U' by auto
  then have cdclW-merge-cp** S U' using ⟨cdclW-merge-cp** S U⟩ by auto
  then show ?thesis using ⟨cdclW-merge-cp** S U⟩ confl' by auto
qed
next
case (bj'-without-decide U') note full-bj = this(1) and cp = this(3)
then have conflicting U ≠ C-True
  using full-bj unfolding full1-def by (fastforce dest!: tranclpD simp: cdclW-bj.simps)
with IH obtain T where
  S-T: cdclW-merge-cp** S T and T-U: conflict T U
  using full-bj unfolding full1-def by (blast dest: tranclpD)
then have cdclW-merge-cp T U'
  using cdclW-merge-cp.conflict'[of T U U'] full-bj by (simp add: full-unfold)
then have S-U': cdclW-merge-cp** S U' using S-T by auto
consider
  (n-s) U' = V
  | (propa) propagate++ U' V
  | (confl') conflict U' V
  | (propa-confl') U'' where propagate++ U' U'' conflict U'' V
  using tranclp-cdclW-cp-propagate-with-conflict-or-not cp
  unfolding rtranclp-unfold full-def by metis
then show ?thesis

```

```

proof cases
  case propa
    then have  $cdcl_W\text{-merge-cp } U' V$  by auto
    moreover have  $conflicting V = C\text{-True}$ 
      using propa unfolding  $trancp\text{-unfold-end}$  by auto
    ultimately show  $?thesis$  using  $S\text{-}U'$  by force
  next
    case  $confl'$ 
    then show  $?thesis$  using  $S\text{-}U'$  by auto
  next
    case  $propa\text{-}confl'$  note  $propa = this(1)$  and  $confl = this(2)$ 
    have  $cdcl_W\text{-merge-cp } U' U''$  using propa by auto
    then show  $?thesis$  using  $S\text{-}U'$   $confl$  by (meson  $rtrancp.rtranc\text{-into-rtranc}$ )
  next
    case  $n\text{-s}$ 
    then show  $?thesis$ 
      using  $S\text{-}U'$  apply (cases  $conflicting V = C\text{-True}$ )
      using  $full\text{-bj}$  apply simp
      by (metis  $cp\ full\text{-def full-unfold full-bj}$ )
qed
qed
qed

lemma  $no\text{-step-cdcl}_W\text{-s}'\text{-no-ste-cdcl}_W\text{-merge-cp}$ :
assumes
   $cdcl_W\text{-all-struct-inv } S$ 
   $conflicting S = C\text{-True}$ 
   $no\text{-step } cdcl_W\text{-s}' S$ 
shows  $no\text{-step } cdcl_W\text{-merge-cp } S$ 
using  $assms$  apply (auto simp:  $cdcl_W\text{-s}'.sims\ cdcl_W\text{-merge-cp}.sims$ )
  using  $conflict\text{-is-full1-cdcl}_W\text{-cp}$  apply blast
using  $cdcl_W\text{-cp-normalized-element-all-inv } cdcl_W\text{-cp.propagate'}$  by (metis  $cdcl_W\text{-cp.propagate'}$ 
   $full\text{-unfold } trancpD$ )

```

The $no\text{-step } decide S$ is needed, since $cdcl_W\text{-merge-cp}$ is $cdcl_W\text{-s}'$ without $decide$.

lemma $conflicting\text{-true-no-step-cdcl}_W\text{-merge-cp-no-step-s}'\text{-without-decide}$:

```

assumes
   $confl$ :  $conflicting S = C\text{-True}$  and
   $inv$ :  $cdcl_W\text{-M-level-inv } S$  and
   $n\text{-s}$ :  $no\text{-step } cdcl_W\text{-merge-cp } S$ 
shows  $no\text{-step } cdcl_W\text{-s}'\text{-without-decide } S$ 
proof (rule  $ccontr$ )
  assume  $\neg no\text{-step } cdcl_W\text{-s}'\text{-without-decide } S$ 
  then obtain  $T$  where
     $cdcl_W$ :  $cdcl_W\text{-s}'\text{-without-decide } S T$ 
    by auto
  then have  $inv\text{-}T$ :  $cdcl_W\text{-M-level-inv } T$ 
    using  $rtrancp\text{-cdcl}_W\text{-s}'\text{-without-decide-rtrancp-cdcl}_W[of S T]$ 
     $rtrancp\text{-cdcl}_W\text{-consistent-inv } inv$  by blast
  from  $cdcl_W$  show  $False$ 
  proof cases
    case  $conflict'\text{-without-decide}$ 
    have  $no\text{-step } propagate S$ 
      using  $n\text{-s}$  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_W-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 ¬ ?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_W-merge-cp-no-step-cdcl_W-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-rtranclp-cdcl_W-merge-cp-no-step-cdcl_W-bj:*

```

  assumes
    conflicting S = C-True and
    cdclW-merge-cp** S T
  shows no-step cdclW-bj T
  using assms(2,1) by (induction)
  (fastforce simp: cdclW-merge-cp.simps full-def tranclp-unfold-end cdclW-bj.simps)+

```


lemma *conflicting-true-full-cdcl_W-merge-cp-iff-full-cdcl_W-s'-without-decode:*

assumes

confl: *conflicting S = C-True* **and**

inv: *cdcl_W-all-struct-inv S*

shows

full cdcl_W-merge-cp S V \longleftrightarrow *full cdcl_W-s'-without-decide S V* (**is** *?fw* \longleftrightarrow *?s'*)

proof

assume *?fw*

then have *st*: *cdcl_W-merge-cp** S V* **and** *n-s*: *no-step cdcl_W-merge-cp V*

unfolding *full-def* **by** *blast+*

have *inv-V*: *cdcl_W-all-struct-inv V*

using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W[of S V]* *(?fw)* **unfolding** *full-def*

by (*simp add: inv rtranclp-cdcl_W-all-struct-inv-inv*)

consider

 (*s'*) *cdcl_W-s'-without-decide** S V*

 | (*propa*) *T* **where** *cdcl_W-s'-without-decide** S T* **and** *propagate⁺⁺ T V*

 | (*bj*) *T U* **where** *cdcl_W-s'-without-decide** S T* **and** *full1 cdcl_W-bj T U* **and** *propagate** U V*

using *rtranclp-cdcl_W-merge-cp-is-rtranclp-cdcl_W-s'-without-decide confl st n-s* **by** *metis*

then have *cdcl_W-s'-without-decide** S V*

proof cases

case *s'*

then show *?thesis* .

next

case *propa* **note** *s' = this(1)* **and** *propa = this(2)*

have *no-step cdcl_W-cp V*

using *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp n-s inv-V*

unfolding *cdcl_W-all-struct-inv-def* **by** *blast*

then have *full1 cdcl_W-cp T V*

using *propa tranclp-mono[of propagate cdcl_W-cp] cdcl_W-cp.propagate'* **unfolding** *full1-def*

by *blast*

then have *cdcl_W-s'-without-decide T V*

using *conflict'-without-decide* **by** *blast*

then show *?thesis* **using** *s'* **by** *auto*

next

case *bj* **note** *s' = this(1)* **and** *bj = this(2)* **and** *propa = this(3)*

have *no-step cdcl_W-cp V*

using *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp n-s inv-V*

unfolding *cdcl_W-all-struct-inv-def* **by** *blast*

then have *full cdcl_W-cp U V*

using *propa rtranclp-mono[of propagate cdcl_W-cp] cdcl_W-cp.propagate'* **unfolding** *full-def*

by *blast*

moreover have *no-step cdcl_W-cp T*

using *bj* **unfolding** *full1-def* **by** (*fastforce dest!: tranclpD simp:cdcl_W-bj.simps*)

ultimately have *cdcl_W-s'-without-decide T V*

using *bj'-without-decide[of T U V]* *bj* **by** *blast*

then show *?thesis* **using** *s'* **by** *auto*

qed

moreover have *no-step cdcl_W-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 cdcl_W-s'.cases*)

```

have ff2: (∀ p s sa. ¬ full1 p (s::'st) sa ∨ p++ s sa ∧ no-step p sa)
  ∧ (∀ p s sa. (¬ p++ (s::'st) sa ∨ (∃ s. p sa s)) ∨ full1 p s sa)
  by (meson full1-def)
obtain ssa :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
  ff3: ∀ p s sa. ¬ p++ s sa ∨ p s (ssa p s sa) ∧ p** (ssa p s sa) sa
  by (metis (no-types) tranclpD)
then have a3: ¬ cdclW-cp++ V ss
  using False by (metis conflicting-clause-full-cdclW-cp full-def)
have ∧s. ¬ cdclW-bj++ V s
  using ff3 False by (metis confl st
    conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj)
then have ¬ cdclW-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 ¬ ?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 tranclp-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 rtranclp-cdclW-all-struct-inv-inv tranclp-into-rtranclp)
    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) $cdcl_W\text{-merge-cp}^{**} S V$ **and** $\text{conflicting } V = C\text{-True}$
 | (cp-false) $cdcl_W\text{-merge-cp}^{**} S V$ **and** $\text{conflicting } V \neq C\text{-True}$ **and** $\text{no-step } cdcl_W\text{-cp } V$ **and**
 $\text{no-step } cdcl_W\text{-bj } V$
 | (cp-conf) T **where** $cdcl_W\text{-merge-cp}^{**} S T$ $\text{conflict } T V$
using $\text{rtrancp-cdcl}_W\text{-s'-without-decide-is-rtrancp-cdcl}_W\text{-merge-cp}[of S V]$ confl
unfolding full-def **by** blast
then have $cdcl_W\text{-merge-cp}^{**} S V$
proof cases
 case cp-conf **note** $S\text{-}T = \text{this}(1)$ **and** $\text{conf-}V = \text{this}(2)$
 have $\text{full } cdcl_W\text{-bj } V V$
 using $\text{conf-}V$ n-s-bj **unfolding** full-def **by** fast
 then have $cdcl_W\text{-merge-cp } T V$
 using $cdcl_W\text{-merge-cp.conflict' conf-}V$ **by** auto
 then show $?thesis$ **using** $S\text{-}T$ **by** auto
qed fast+
moreover
then have $cdcl_W^{**} S V$ **using** $\text{rtrancp-cdcl}_W\text{-merge-cp-rtrancp-cdcl}_W$ **by** blast
then have $cdcl_W\text{-all-struct-inv } V$
 using $\text{inv rtrancp-cdcl}_W\text{-all-struct-inv-inv}$ **by** blast
then have $\text{no-step } cdcl_W\text{-merge-cp } V$
 using $\text{conflicting-true-no-step-s'-without-decide-no-step-cdcl}_W\text{-merge-cp } s'$
 unfolding full-def **by** blast
ultimately show $?fw$ **unfolding** full-def **by** auto
qed

lemma $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-iff-full1-cdcl}_W\text{-s'-without-decode}$:
assumes
 $\text{confl: conflicting } S = C\text{-True}$ **and**
 $\text{inv: cdcl}_W\text{-all-struct-inv } S$
shows
 $\text{full1 } cdcl_W\text{-merge-cp } S V \longleftrightarrow \text{full1 } cdcl_W\text{-s'-without-decide } S V$
proof –
have $\text{full } cdcl_W\text{-merge-cp } S V = \text{full } cdcl_W\text{-s'-without-decide } S V$
 using $\text{confl conflicting-true-full-cdcl}_W\text{-merge-cp-iff-full-cdcl}_W\text{-s'-without-decide inv}$
 by blast
then show $?thesis$ **unfolding** $\text{full-unfold full1-def}$
 by $(metis (\text{mono-tags}) \text{trancp-unfold-begin})$
qed

lemma $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-imp-full1-cdcl}_W\text{-s'-without-decode}$:
assumes
 $\text{fw: full1 } cdcl_W\text{-merge-cp } S V$ **and**
 $\text{inv: cdcl}_W\text{-all-struct-inv } S$
shows
 $\text{full1 } cdcl_W\text{-s'-without-decide } S V$
proof –
have $\text{conflicting } S = C\text{-True}$
 using $\text{fw unfolding full1-def}$ **by** $(\text{auto dest!: trancpD simp: cdcl}_W\text{-merge-cp.simps})$
then show $?thesis$
 using $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-iff-full1-cdcl}_W\text{-s'-without-decode fw inv}$ **by** blast
qed

inductive $cdcl_W\text{-merge-stgy}$ **where**
 $\text{fw-s-cp}[\text{intro}]: \text{full1 } cdcl_W\text{-merge-cp } S T \implies cdcl_W\text{-merge-stgy } S T$ |

fw-s-decide[intro]: $\text{decide } S \ T \implies \text{no-step } \text{cdcl}_W\text{-merge-cp } S \implies \text{full } \text{cdcl}_W\text{-merge-cp } T \ U$
 $\implies \text{cdcl}_W\text{-merge-stgy } S \ U$

lemma *cdcl_W-merge-stgy-tranclp-cdcl_W-merge*:

assumes *fw*: *cdcl_W-merge-stgy* *S T*

shows *cdcl_W-merge⁺⁺* *S T*

proof –

{ **fix** *S T*

assume *full1 cdcl_W-merge-cp* *S T*

then have *cdcl_W-merge⁺⁺* *S T*

using *tranclp-mono*[of *cdcl_W-merge-cp cdcl_W-merge⁺⁺*] *cdcl_W-merge-cp-tranclp-cdcl_W-merge*

unfolding *full1-def*

by *auto*

} **note** *full1-cdcl_W-merge-cp-cdcl_W-merge* = *this*

show *?thesis*

using *fw*

apply (*induction rule: cdcl_W-merge-stgy.induct*)

using *full1-cdcl_W-merge-cp-cdcl_W-merge* **apply** *simp*

unfolding *full-unfold* **by** (*auto dest!: full1-cdcl_W-merge-cp-cdcl_W-merge fw-decide*)

qed

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-merge*:

assumes *fw*: *cdcl_W-merge-stgy^{**}* *S T*

shows *cdcl_W-merge^{**}* *S T*

using *fw cdcl_W-merge-stgy-tranclp-cdcl_W-merge rtranclp-mono*[of *cdcl_W-merge-stgy cdcl_W-merge⁺⁺*]

unfolding *tranclp-rtranclp-rtranclp* **by** *blast*

lemma *cdcl_W-merge-stgy-rtranclp-cdcl_W*:

cdcl_W-merge-stgy *S T* \implies *cdcl_W^{**}* *S T*

apply (*induction rule: cdcl_W-merge-stgy.induct*)

using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W* **unfolding** *full1-def*

apply (*simp add: tranclp-into-rtranclp*)

using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W cdcl_W-o.decide cdcl_W.other* **unfolding** *full-def*

by (*meson r-into-rtranclp rtranclp-trans*)

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W*:

*cdcl_W-merge-stgy^{**}* *S T* \implies *cdcl_W^{**}* *S T*

using *rtranclp-mono*[of *cdcl_W-merge-stgy cdcl_W^{**}*] *cdcl_W-merge-stgy-rtranclp-cdcl_W* **by** *auto*

lemma *cdcl_W-merge-stgy-cases*[*consumes 1, case-names fw-s-cp fw-s-decide*]:

assumes

cdcl_W-merge-stgy *S U*

full1 cdcl_W-merge-cp *S U* \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_W-merge-stgy.simps*)

inductive *cdcl_W-s'-w* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**

conflict': *full1 cdcl_W-s'-without-decide* *S S'* \implies *cdcl_W-s'-w* *S S'* |

decide': *decide* *S S'* \implies *no-step cdcl_W-s'-without-decide* *S* \implies *full cdcl_W-s'-without-decide* *S' S''*
 \implies *cdcl_W-s'-w* *S S''*

lemma *cdcl_W-s'-w-rtranclp-cdcl_W*:

cdcl_W-s'-w *S T* \implies *cdcl_W^{**}* *S T*

apply (*induction rule: cdcl_W-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 $rtrancp-cdcl_W-s'-without-decide-rtrancp-cdcl_W\ rtrancp-cdcl_W-all-struct-inv-inv$
 $no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj$ **unfolding** *full1-def*
apply (*meson trancp-into-rtrancp*)
using $rtrancp-cdcl_W-s'-without-decide-rtrancp-cdcl_W\ rtrancp-cdcl_W-all-struct-inv-inv$
 $no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj$ **unfolding** *full-def*
by (*meson cdcl_W-merge-restart-cdcl_W fw-r-decide*)

lemma $rtrancp-cdcl_W-s'-w-no-step-cdcl_W-bj-or-eq$:
assumes $cdcl_W-s'-w^{**}\ S\ T$ **and** $cdcl_W-all-struct-inv\ S$
shows $S = T \vee no-step\ cdcl_W-bj\ T$
using *assms apply induction*
apply *simp*
using $rtrancp-cdcl_W-s'-w-rtrancp-cdcl_W\ rtrancp-cdcl_W-all-struct-inv-inv$
 $cdcl_W-s'-w-no-step-cdcl_W-bj$ **by** *meson*

lemma $rtrancp-cdcl_W-s'-no-step-cdcl_W-s'-without-decide-decomp-into-cdcl_W-merge$:
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  $W = C-True$ 
    | (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 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  $T-V = this(4)$ 
  consider
    (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  $V W$ ] 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  $W = C-True$ 
    | (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.prem)
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 decide-rtrancpl-cdclW-s'-rtrancpl-cdclW-s':
  assumes
    dec: decide S T and
    cdclW-s'** T U and
    n-s-S: no-step cdclW-cp S and
    no-step cdclW-cp U
  shows cdclW-s'** 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 cdclW-s' T T' and cdclW-s'** T' U
  using st[unfolded rtrancpl-unfold] by (auto dest!: trancplD)
  then show ?case
  proof cases
    case TU
    then show ?thesis
    proof -
      assume a1: T = U
      then have f2: cdclW-s' T V
      using s' by force
      obtain ss :: 'st where
        cdclW-s'** S T ∨ cdclW-cp T ss
      using a1 step.IH by blast
      then show ?thesis
      using f2 by (metis (full-types) cdclW-s'.decide' cdclW-s'E dec full1-is-full n-s-S
        rtrancpl-unfold trancpl-unfold-end)
    qed
  next
    case (s'-st T') note s'-T' = this(1) and st = this(2)
    have cdclW-s'** S T'
    using s'-T'
    proof cases
      case conflict'
      then have cdclW-s' S T'
      using dec cdclW-s'.decide' n-s-S by (simp add: full-unfold)
      then show ?thesis
      using st by auto
    next
      case (decide' T'')
      then have cdclW-s' S T
      using dec cdclW-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!: trancplD simp: cdclW-bj.simps)
      then show ?thesis by fast
    qed
  then show ?thesis using s' st by auto

```

```

    qed
next
case base
then have full cdclW-cp T T
  by (simp add: full-unfold)
then show ?case
  using cdclW-s'.simps dec n-s-S by auto
qed

lemma rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s':
  assumes
    cdclW-merge-stgy** R V and
    inv: cdclW-all-struct-inv R
  shows cdclW-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 cdclW-all-struct-inv S
  using inv rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW st by blast
from fw show ?case
proof (cases rule: cdclW-merge-stgy-cases)
case fw-s-cp
then show ?thesis
proof -
  assume a1: full1 cdclW-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: cdclW-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:  $\bigwedge s. \neg full1\ cdcl_W\text{-merge-cp}\ s\ S$ 
    using f3 f2 by (metis (full-types))
  have  $\bigwedge s. \neg full\ cdcl_W\text{-merge-cp}\ s\ S$ 
    using f4 f3 by (meson full-def)
  then have S = R
    using f5 by (metis (no-types) cdclW-merge-stgy.simps rtranclp-unfold st tranclp-unfold-end)
  then show ?thesis
    using f2 a1 by (metis (no-types)  $\langle cdcl_W\text{-all-struct-inv}\ S \rangle$ 
      conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode
      rtranclp-cdclW-s'-without-decide-rtranclp-cdclW-s' 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 conflicting S' = C-True
  by auto
ultimately have full cdclW-s'-without-decide S' T

```

```

    by (meson ⟨cdclW-all-struct-inv S⟩ cdclW-merge-restart-cdclW fw-r-decide
        rtrancpl-cdclW-all-struct-inv-inv
        conflicting-true-full-cdclW-merge-cp-iff-full-cdclW-s'-without-decode)
  then have a1: cdclW-s*** S' T
    unfolding full-def by (metis (full-types)rtrancpl-cdclW-s'-without-decide-rtrancpl-cdclW-s')
  have cdclW-merge-stgy** S T
    using fw by blast
  then have cdclW-s*** S T
    using decide-rtrancpl-cdclW-s'-rtrancpl-cdclW-s' a1 by (metis ⟨cdclW-all-struct-inv S⟩ dec
        n-S no-step-cdclW-merge-cp-no-step-cdclW-cp cdclW-all-struct-inv-def
        rtrancpl-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
  then show ?thesis using IH by auto
qed
qed

```

lemma *rtrancpl-cdcl_W-merge-stgy-distinct-mset-clauses:*

```

  assumes invR: cdclW-all-struct-inv R and
    st: cdclW-merge-stgy** R S and
    dist: distinct-mset (clauses R) and
    R: trail R = []
  shows distinct-mset (clauses S)
  using rtrancpl-cdclW-stgy-distinct-mset-clauses[OF invR - dist R]
  invR st rtrancpl-mono[of cdclW-s' cdclW-stgy**] cdclW-s'-is-rtrancpl-cdclW-stgy
  by (auto dest!: cdclW-s'-is-rtrancpl-cdclW-stgy rtrancpl-cdclW-merge-stgy-rtrancpl-cdclW-s')

```

lemma *no-step-cdcl_W-s'-no-step-cdcl_W-merge-stgy:*

```

  assumes
    inv: cdclW-all-struct-inv R and s': no-step cdclW-s' R
  shows no-step cdclW-merge-stgy R

```

proof –

```

{ fix ss :: 'st
  obtain ssa :: 'st ⇒ 'st ⇒ 'st where
    ff1: ⋀ s sa. ¬ cdclW-merge-stgy s sa ∨ full1 cdclW-merge-cp s sa ∨ decide s (ssa s sa)
    using cdclW-merge-stgy.cases by moura
  obtain ssb :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
    ff2: ⋀ p s sa. ¬ p++ s sa ∨ p s (ssb p s sa)
    by (meson trancpl-unfold-begin)
  obtain ssc :: 'st ⇒ 'st where
    ff3: ⋀ s sa sb. (¬ cdclW-all-struct-inv s ∨ ¬ cdclW-cp s sa ∨ cdclW-s' s (ssc s))
      ∧ (¬ cdclW-all-struct-inv s ∨ ¬ cdclW-o s sb ∨ cdclW-s' s (ssc s))
    using n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o by moura
  then have ff4: ⋀ s. ¬ cdclW-o R s
    using s' inv by blast
  have ff5: ⋀ s. ¬ cdclW-cp++ R s
    using ff3 ff2 s' by (metis inv)
  have ⋀ s. ¬ cdclW-bj++ R s
    using ff4 ff2 by (metis bj)
  then have ⋀ s. ¬ cdclW-s'-without-decide R s
    using ff5 by (simp add: cdclW-s'-without-decide.simps full1-def)
  then have ¬ cdclW-s'-without-decide++ R ss
    using ff2 by blast
  then have ¬ cdclW-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-cdcl_W-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-cdcl_W-merge* **by** (rule *wf-subset*) (auto simp: *cdcl_W-merge-cp-tranclp-cdcl_W-merge*)

lemma *wf-cdcl_W-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-cdcl_W-merge* **by** (rule *wf-subset*)
(auto simp add: *cdcl_W-merge-stgy-tranclp-cdcl_W-merge*)

lemma *cdcl_W-merge-cp-obtain-normal-form*:

assumes *inv*: *cdcl_W-all-struct-inv* *R*
obtains *S* **where** *full cdcl_W-merge-cp* *R* *S*

proof –

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-cdcl_W-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 *cdcl_W-merge-cp*** *R* *S*

using *st* **by** *induction auto*

moreover

have *cdcl_W-all-struct-inv* *S*

using *st inv*

apply (*induction rule*: *rtranclp-induct*)

apply *simp*

by (*meson r-into-rtranclp rtranclp-cdcl_W-all-struct-inv-inv*
rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W)

then have *no-step cdcl_W-merge-cp* *S*

using *n-s* **by** *auto*

ultimately show *?thesis*

using *that unfolding full-def* **by** *blast*

qed

lemma *no-step-cdcl_W-merge-stgy-no-step-cdcl_W-s'*:

assumes

inv: *cdcl_W-all-struct-inv* *R* **and**

confl: *conflicting* *R* = *C-True* **and**

n-s: *no-step cdcl_W-merge-stgy* *R*

shows *no-step cdcl_W-s'* *R*

proof (rule *ccontr*)

assume $\neg ?thesis$

then obtain *S* **where** *cdcl_W-s'* *R* *S* **by** *auto*

then show *False*

proof *cases*

case *conflict'*

then obtain *S'* **where** *full1 cdcl_W-merge-cp* *R* *S'*

by (*metis* (*full-types*) *cdcl_W-merge-cp-obtain-normal-form cdcl_W-s'-without-decide.simps confl*
conflicting-true-no-step-cdcl_W-merge-cp-no-step-s'-without-decide full-def full-unfold inv
cdcl_W-all-struct-inv-def)

then show *False* **using** *n-s* **by** *blast*

next

```

    case (decide' R')
    then have cdclW-all-struct-inv R'
      using inv cdclW-all-struct-inv-inv cdclW.other cdclW-o.decide by meson
    then obtain R'' where full cdclW-merge-cp R' R''
      using cdclW-merge-cp-obtain-normal-form by blast
    moreover have no-step cdclW-merge-cp R
      by (simp add: confl local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart)
    ultimately show False using n-s cdclW-merge-stgy.intros local.decide'(1) by blast
  next
  case (bj' R')
  then show False
    using confl no-step-cdclW-cp-no-step-cdclW-s'-without-decide inv
    unfolding cdclW-all-struct-inv-def by blast
qed
qed

```

```

lemma rtranclp-cdclW-merge-cp-no-step-cdclW-bj:
  assumes conflicting R = C-True and cdclW-merge-cp** R S
  shows no-step cdclW-bj S
  using assms conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj by blast

```

```

lemma rtranclp-cdclW-merge-stgy-no-step-cdclW-bj:
  assumes confl: conflicting R = C-True and cdclW-merge-stgy** R S
  shows no-step cdclW-bj S
  using assms(2)

```

proof induction

```

  case base
  then show ?case
    using confl by (auto simp: cdclW-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 cdclW-merge-cp.simps dest!: tranclpD)
  from fw show ?case
  proof cases
    case fw-s-cp
    then show ?thesis
      using rtranclp-cdclW-merge-cp-no-step-cdclW-bj confl-S
      by (simp add: full1-def tranclp-into-rtranclp)
  next
    case (fw-s-decide S')
    moreover then have conflicting S' = C-True by auto
    ultimately show ?thesis
      using conflicting-not-true-rtranclp-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  $cdcl_W-s'^{**} R V$  unfolding full-def by blast
have  $cdcl_W\text{-all-struct-inv } V$ 
  using  $\langle cdcl_W-s'^{**} R V \rangle$   $inv \text{ rtranclp-}cdcl_W\text{-all-struct-inv-inv rtranclp-}cdcl_W-s'\text{-rtranclp-}cdcl_W$ 
  by blast
then have  $n\text{-s: no-step } cdcl_W\text{-merge-stgy } V$ 
  using  $no\text{-step-}cdcl_W-s'\text{-no-step-}cdcl_W\text{-merge-stgy}$  by ( $meson \langle full \text{ } cdcl_W-s' R V \rangle$  full-def)
have  $n\text{-s-bj: no-step } cdcl_W\text{-bj } V$ 
  by ( $metis \langle cdcl_W\text{-all-struct-inv } V \rangle \langle full \text{ } cdcl_W-s' R V \rangle$  bj full-def
     $n\text{-step-}cdcl_W\text{-stgy-iff-no-step-}cdcl_W\text{-cl-}cdcl_W\text{-o}$ )
have  $n\text{-s-cp: no-step } cdcl_W\text{-merge-cp } V$ 
proof –
  { fix  $ss :: 'st$ 
    obtain  $ssa :: 'st \Rightarrow 'st$  where
       $ff1: \forall s. \neg cdcl_W\text{-all-struct-inv } s \vee cdcl_W-s'\text{-without-decide } s (ssa \ s)$ 
       $\vee no\text{-step } cdcl_W\text{-merge-cp } s$ 
      using  $conflicting\text{-true-no-step-}s'\text{-without-decide-no-step-}cdcl_W\text{-merge-cp}$  by moura
      have  $(\forall p \ s \ sa. \neg full \ p \ (s::'st) \ sa \vee p^{**} \ s \ sa \wedge no\text{-step } p \ sa)$  and
         $(\forall p \ s \ sa. (\neg p^{**} \ (s::'st) \ sa \vee (\exists s. p \ sa \ s)) \vee full \ p \ s \ sa)$ 
        by ( $meson$  full-def)+
      then have  $\neg cdcl_W\text{-merge-cp } V \ ss$ 
        using  $ff1$  by ( $metis (no\text{-types}) \langle cdcl_W\text{-all-struct-inv } V \rangle \langle full \text{ } cdcl_W-s' R V \rangle$   $cdcl_W-s'\text{-simps}$ 
           $cdcl_W-s'\text{-without-decide.cases}$ ) }
      then show ?thesis
        by blast
    }
  qed
consider
  ( $fw\text{-no-confl}$ )  $cdcl_W\text{-merge-stgy}^{**} R V$  and  $conflicting \ V = C\text{-True}$ 
| ( $fw\text{-confl}$ )  $cdcl_W\text{-merge-stgy}^{**} R V$  and  $conflicting \ V \neq C\text{-True}$  and  $no\text{-step } cdcl_W\text{-bj } V$ 
| ( $fw\text{-dec-confl}$ )  $S \ T \ U$  where  $cdcl_W\text{-merge-stgy}^{**} R S$  and  $no\text{-step } cdcl_W\text{-merge-cp } S$  and
   $decide \ S \ T$  and  $cdcl_W\text{-merge-cp}^{**} T \ U$  and  $conflict \ U \ V$ 
| ( $fw\text{-dec-no-confl}$ )  $S \ T$  where  $cdcl_W\text{-merge-stgy}^{**} R S$  and  $no\text{-step } cdcl_W\text{-merge-cp } S$  and
   $decide \ S \ T$  and  $cdcl_W\text{-merge-cp}^{**} T \ V$  and  $conflicting \ V = C\text{-True}$ 
| ( $cp\text{-no-confl}$ )  $cdcl_W\text{-merge-cp}^{**} R V$  and  $conflicting \ V = C\text{-True}$ 
| ( $cp\text{-confl}$ )  $U$  where  $cdcl_W\text{-merge-cp}^{**} R U$  and  $conflict \ U \ V$ 
using  $rtranclp\text{-}cdcl_W-s'\text{-no-step-}cdcl_W-s'\text{-without-decide-decomp-into-}cdcl_W\text{-merge}[OF$ 
   $\langle cdcl_W-s'^{**} R V \rangle$   $assms]$  by auto
then show ?fw
proof cases
  case  $fw\text{-no-confl}$ 
    then show ?thesis using  $n\text{-s}$  unfolding full-def by blast
  next
    case  $fw\text{-confl}$ 
      then show ?thesis using  $n\text{-s}$  unfolding full-def by blast
  next
    case  $fw\text{-dec-confl}$ 
      have  $cdcl_W\text{-merge-cp } U \ V$ 
        using  $n\text{-s-bj}$  by ( $metis \text{ } cdcl_W\text{-merge-cp.simps full-unfold } fw\text{-dec-confl}(5)$ )
      then have  $full1 \ cdcl_W\text{-merge-cp } T \ V$ 
        unfolding full1-def by ( $metis \text{ } fw\text{-dec-confl}(4) \ n\text{-s-cp } tranclp\text{-unfold-end}$ )
      then have  $cdcl_W\text{-merge-stgy } S \ V$  using  $\langle decide \ S \ T \rangle \langle no\text{-step } cdcl_W\text{-merge-cp } S \rangle$  by auto
      then show ?thesis using  $n\text{-s}$   $\langle cdcl_W\text{-merge-stgy}^{**} R S \rangle$  unfolding full-def by auto
  next
    case  $fw\text{-dec-no-confl}$ 
      then have  $full \ cdcl_W\text{-merge-cp } T \ V$ 

```

```

    using n-s-cp unfolding full-def by blast
  then have cdclW-merge-stgy S V using  $\langle \text{decide } S \ T \rangle \langle \text{no-step } \text{cdcl}_W\text{-merge-cp } S \rangle$  by auto
  then show ?thesis using n-s  $\langle \text{cdcl}_W\text{-merge-stgy}^{**} R \ S \rangle$  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  $R = V \vee \text{cdcl}_W\text{-merge-stgy}^{++} R \ V$ 
  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  $\langle ?fw \rangle$  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-cdclW-stgy-full-cdclW-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-cdclW-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

```

19.6 Adding Restarts

```

locale cdclW-ops-restart =
  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  $\Rightarrow$  ('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-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  add-learned-cls 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::linorder clauses  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st +
fixes f :: nat  $\Rightarrow$  nat
assumes f: unbounded f
begin

```

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive *cdcl_W-merge-with-restart* **where**

```

restart-step:
  (cdclW-merge-stgy  $\sim$  (card (set-mset (learned-clss T)) – card (set-mset (learned-clss S)))) S T
 $\implies$  card (set-mset (learned-clss T)) – card (set-mset (learned-clss S)) > f n
 $\implies$  restart T U  $\implies$  cdclW-merge-with-restart (S, n) (U, Suc n) |
restart-full: full1 cdclW-merge-stgy S T  $\implies$  cdclW-merge-with-restart (S, n) (T, Suc n)

```

lemma *cdcl_W-merge-with-restart S T* \implies *cdcl_W-merge-restart*** (*fst S*) (*fst T*)

```

by (induction rule: cdclW-merge-with-restart.induct)
  (auto dest!: relpow-imp-rtranclp cdclW-merge-stgy-tranclp-cdclW-merge tranclp-into-rtranclp)

```

rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W-merge *rtrancpl-cdcl_W-merge-trancpl-cdcl_W-merge-restart*
fw-r-rf cdcl_W-rf.restart
simp: full1-def)

lemma *cdcl_W-merge-with-restart-rtrancpl-cdcl_W:*
*cdcl_W-merge-with-restart S T \implies cdcl_W** (fst S) (fst T)*
by (*induction rule: cdcl_W-merge-with-restart.induct*)
(auto dest!: relpowp-imp-rtrancpl rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W cdcl_W.rf
cdcl_W-rf.restart trancpl-into-rtrancpl simp: full1-def)

lemma *cdcl_W-merge-with-restart-increasing-number:*
cdcl_W-merge-with-restart S T \implies snd T = 1 + snd S
by (*induction rule: cdcl_W-merge-with-restart.induct*) *auto*

lemma *full1 cdcl_W-merge-stgy S T \implies cdcl_W-merge-with-restart (S, n) (T, Suc n)*
using *restart-full* **by** *blast*

lemma *cdcl_W-all-struct-inv-learned-clss-bound:*
assumes *inv: cdcl_W-all-struct-inv S*
shows *set-mset (learned-clss S) \subseteq build-all-simple-clss (atms-of-msu (init-clss S))*

proof

fix *C*

assume *C: C \in set-mset (learned-clss S)*

have *distinct-mset C*

using *C inv unfolding cdcl_W-all-struct-inv-def distinct-cdcl_W-state-def distinct-mset-set-def*
by *auto*

moreover **have** \neg *tautology C*

using *C inv unfolding cdcl_W-all-struct-inv-def cdcl_W-learned-clause-def* **by** *auto*

moreover

have *atms-of C \subseteq atms-of-msu (learned-clss S)*

using *C* **by** *auto*

then **have** *atms-of C \subseteq atms-of-msu (init-clss S)*

using *inv unfolding cdcl_W-all-struct-inv-def no-strange-atm-def* **by** *force*

moreover **have** *finite (atms-of-msu (init-clss S))*

using *inv unfolding cdcl_W-all-struct-inv-def* **by** *auto*

ultimately **show** *C \in build-all-simple-clss (atms-of-msu (init-clss S))*

using *distinct-mset-not-tautology-implies-in-build-all-simple-clss build-all-simple-clss-mono*
by *blast*

qed

lemma *cdcl_W-merge-with-restart-init-clss:*
cdcl_W-merge-with-restart S T \implies cdcl_W-M-level-inv (fst S) \implies
init-clss (fst S) = init-clss (fst T)
using *cdcl_W-merge-with-restart-rtrancpl-cdcl_W rtrancpl-cdcl_W-init-clss* **by** *blast*

lemma

wf {(T, S). cdcl_W-all-struct-inv (fst S) \wedge cdcl_W-merge-with-restart S T}

proof (*rule ccontr*)

assume \neg *?thesis*

then **obtain** *g* **where**

g: $\bigwedge i. cdcl_W-merge-with-restart (g i) (g (Suc i))$ and

inv: $\bigwedge i. cdcl_W-all-struct-inv (fst (g i))$

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-merge-with-restart-init-clss)
  } note init-g = this
let ?S = g 0
have finite (atms-of-msu (init-clss (fst ?S)))
  using inv unfolding cdclW-all-struct-inv-def by auto
have snd-g:  $\bigwedge i. \text{snd } (g \ i) = i + \text{snd } (g \ 0)$ 
  apply (induct-tac i)
  apply simp
  by (metis Suc-eq-plus1-left add-Suc cdclW-merge-with-restart-increasing-number g)
then have snd-g-0:  $\bigwedge i. i > 0 \implies \text{snd } (g \ i) = i + \text{snd } (g \ 0)$ 
  by blast
have unbounded-f-g: unbounded ( $\lambda i. f \ (\text{snd } (g \ i))$ )
  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:  $f \ (\text{snd } (g \ k)) > \text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$  and
  k >  $\text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$ 
  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-merge-stgy (fst (g i))
  with g[of i]
  have False
    proof (induction rule: cdclW-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 cdclW-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: tranclpD)
    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
  restart T (fst (g (k+1))) and
  cdclW-merge-stgy:  $(\text{cdcl}_W\text{-merge-stgy } \widetilde{\sim} m) \ (\text{fst } (g \ k)) \ T$ 
  using g[of k] H[of Suc k] by (force simp: cdclW-merge-with-restart.simps full1-def)
have cdclW-merge-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-merge-stgy-rtranclp-cdclW
  by blast
moreover have  $\text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } (\text{fst } (g \ k))))$ 
  >  $\text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$ 
  unfolding m[symmetric] using m > f (snd (g k)) f-g-k by linarith
then have  $\text{card } (\text{set-mset } (\text{learned-clss } T))$ 
  >  $\text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$ 
  by linarith
moreover

```

```

have init-clss (fst (g k)) = init-clss T
  using ⟨cdclW-merge-stgy** (fst (g k)) T⟩ rtrancp-cdclW-merge-stgy-rtrancp-cdclW
  rtrancp-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-merge-with-restart-distinct-mset-clauses:
  assumes invR: cdclW-all-struct-inv (fst R) and
  st: cdclW-merge-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 rtrancp-cdclW-merge-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: trancp-into-rtrancp)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T)
    using rtrancp-cdclW-merge-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: relpowp-imp-rtrancp)
  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

inductive cdclW-with-restart where
  restart-step:
    (cdclW-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 ⇒
      cdclW-with-restart (S, n) (U, Suc n) |
  restart-full: full1 cdclW-stgy S T ⇒ cdclW-with-restart (S, n) (T, Suc n)

lemma cdclW-with-restart-rtrancp-cdclW:
  cdclW-with-restart S T ⇒ cdclW** (fst S) (fst T)
  apply (induction rule: cdclW-with-restart.induct)
  by (auto dest!: relpowp-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 ⇒ snd T = 1 + snd S
  by (induction rule: cdclW-with-restart.induct) auto

lemma full1 cdclW-stgy S T ⇒ cdclW-with-restart (S, n) (T, Suc n)
  using restart-full by blast

lemma cdclW-with-restart-init-clss:
  cdclW-with-restart S T ⇒ cdclW-M-level-inv (fst S) ⇒ init-clss (fst S) = init-clss (fst T)
  using cdclW-with-restart-rtrancp-cdclW rtrancp-cdclW-init-clss by blast

```


lemma

wf $\{(T, S). \text{cdcl}_W\text{-all-struct-inv } (\text{fst } S) \wedge \text{cdcl}_W\text{-with-restart } S \ T\}$

proof (*rule ccontr*)

assume $\neg \text{?thesis}$

then obtain *g* **where**

g: $\bigwedge i. \text{cdcl}_W\text{-with-restart } (g \ i) \ (g \ (\text{Suc } i))$ **and**

inv: $\bigwedge i. \text{cdcl}_W\text{-all-struct-inv } (\text{fst } (g \ i))$

unfolding *wf-iff-no-infinite-down-chain* **by** *fast*

{ fix *i*

have *init-clss* $(\text{fst } (g \ i)) = \text{init-clss } (\text{fst } (g \ 0))$

apply (*induction i*)

apply *simp*

using *g inv unfolding cdcl_W-all-struct-inv-def* **by** (*metis cdcl_W-with-restart-init-clss*)

} **note** *init-g = this*

let *?S = g 0*

have *finite* (*atms-of-msu* (*init-clss* (*fst ?S*)))

using *inv unfolding cdcl_W-all-struct-inv-def* **by** *auto*

have *snd-g*: $\bigwedge i. \text{snd } (g \ i) = i + \text{snd } (g \ 0)$

apply (*induct-tac i*)

apply *simp*

by (*metis Suc-eq-plus1-left add-Suc cdcl_W-with-restart-increasing-number g*)

then have *snd-g-0*: $\bigwedge i. i > 0 \implies \text{snd } (g \ i) = i + \text{snd } (g \ 0)$

by *blast*

have *unbounded-f-g*: *unbounded* $(\lambda i. f \ (\text{snd } (g \ i)))$

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: $f \ (\text{snd } (g \ k)) > \text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$ **and**

$k > \text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$

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_W-stgy* (*fst* (*g i*))

with *g[of i]*

have *False*

proof (*induction rule: cdcl_W-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_W-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: $m = \text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } (\text{fst } (g \ k))))$ **and**

$m > f \ (\text{snd } (g \ k))$ **and**

restart T (*fst* (*g (k+1)*)) **and**

cdcl_W-merge-stgy: (*cdcl_W-stgy* $\widetilde{\sim} m$) (*fst* (*g k*)) *T*

using *g[of k] H[of Suc k]* **by** (*force simp: cdcl_W-with-restart.simps full1-def*)

have *cdcl_W-stgy*** (*fst* (*g k*)) *T*

```

    using cdclW-merge-stgy relpoup-imp-rtrancp by metis
  then have cdclW-all-struct-inv T
    using inv[of k] rtrancp-cdclW-all-struct-inv-inv rtrancp-cdclW-stgy-rtrancp-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-msu (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-msu (init-clss (fst ?S))))
    by linarith
  moreover
  have init-clss (fst (g k)) = init-clss T
    using ⟨cdclW-stgy** (fst (g k)) T⟩ rtrancp-cdclW-stgy-rtrancp-cdclW rtrancp-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 rtrancp-cdclW-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: trancp-into-rtrancp)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T) using rtrancp-cdclW-stgy-distinct-mset-clauses[of S T]
    unfolding full1-def by (auto dest: relpoup-imp-rtrancp)
  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  $\exists k :: \text{nat}. (2 \wedge (k - 1) \leq i \wedge i < 2 \wedge k - 1) \vee i = 2 \wedge k - 1$ 
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 ⇒ nat where
luby-sequence-core i =
  (if ∃ k. i = 2^k - 1
   then 2^((SOME k. i = 2^k - 1) - 1)
   else luby-sequence-core (i - 2^((SOME k. 2^(k-1) ≤ i ∧ i < 2^k - 1) - 1) + 1))
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^{k-1} ≤ i ∧ i < 2^k - 1)
  have 2^{?k-1} ≤ i ∧ i < 2^{?k} - 1
    apply (rule someI-ex)
    using 2 exists-luby-decomp by blast
  then show ?case

proof -
  have ∀ n na. ¬ (1::nat) ≤ n ∨ 1 ≤ n ∧ na
    by (meson one-le-power)
  then have f1: (1::nat) ≤ 2^{?k-1}
    using one-le-numeral by blast
  have f2: i - 2^{?k-1} + 2^{?k-1} = i

```

```

    using (2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1) le-add-diff-inverse2 by blast
have f3: 2 ^ ?k - 1 ≠ Suc 0
    using f1 (2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1) by linarith
have 2 ^ ?k - (1::nat) ≠ 0
    using (2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1) gr-implies-not0 by blast
then have f4: 2 ^ ?k ≠ (1::nat)
    by linarith
have f5: ∀ n na. if na = 0 then (n::nat) ^ na = 1 else n ^ na = n * n ^ (na - 1)
    by (simp add: power-eq-if)
then have ?k ≠ 0
    using f4 by meson
then have 2 ^ (?k - 1) ≠ Suc 0
    using f5 f3 by presburger
then have Suc 0 < 2 ^ (?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::nat) ^ (k::nat) - 1 = 2 ^ k' - 1
  shows k' = k
proof -
  have (2::nat) ^ (k::nat) = 2 ^ 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 ^ k - 1) = 2 ^ (k-1) (is ?L = ?K)
proof -
  have decomp: ∃ ka. 2 ^ k - 1 = 2 ^ ka - 1
    by auto
  have ?L = 2 ^ ((SOME k'. (2::nat) ^ k - 1 = 2 ^ 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: $2 \wedge (k - 1) \leq i$ and

i-k: $i < 2 \wedge k - 1$

shows luby-sequence-core i = luby-sequence-core (i - $2 \wedge (k - 1) + 1$)

proof -

have H: $\neg (\exists ka. i = 2 \wedge ka - 1)$

proof (rule ccontr)

assume $\neg ?thesis$

then obtain k'::nat where k': $i = 2 \wedge k' - 1$ by blast

have $(2::nat) \wedge k' - 1 < 2 \wedge k - 1$

using i-k unfolding k'.

then have $(2::nat) \wedge k' < 2 \wedge k$

by linarith

then have $k' < k$

by simp

have $2 \wedge (k - 1) \leq 2 \wedge k' - (1::nat)$

using k-i unfolding k'.

then have $(2::nat) \wedge (k-1) < 2 \wedge k'$

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 $\langle k' < k \rangle \langle k-1 < k' \rangle$ by linarith

qed

have $\bigwedge k k'. 2 \wedge (k - \text{Suc } 0) \leq i \implies i < 2 \wedge k - \text{Suc } 0 \implies 2 \wedge (k' - \text{Suc } 0) \leq i \implies$
 $i < 2 \wedge k' - \text{Suc } 0 \implies k = k'$

by (meson different-luby-decomposition-false linorder-neqE-nat)

then have k: $(\text{SOME } k. 2 \wedge (k - \text{Suc } 0) \leq i \wedge i < 2 \wedge k - \text{Suc } 0) = k$

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 luby-sequence-core $(2 \wedge (b+1) - 1) = 2 \wedge b$

using luby-sequence-core-two-power-minus-one[of b+1] by simp

moreover have $(2::nat) \wedge b > b$

by (induction b) auto

ultimately show False using b[of $2 \wedge (b+1) - 1$] by linarith

qed

abbreviation luby-sequence :: nat \Rightarrow nat where

luby-sequence n \equiv ur * 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::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 ≥ 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  $\text{Suc } n = 2^k - 1$ 
  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: Suc n - 2^k + 1 < Suc n
            by (metis Suc-1 Suc-eq-plus1 add.commute add-diff-cancel-left' add-less-mono1 gr0I
              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 ⇒ ('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 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::linorder clauses ⇒ 'st and
restart-state :: 'st ⇒ '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_W-all-struct-inv*

```

definition cdclW-stgy-invariant where
cdclW-stgy-invariant S ⇔
  conflict-is-false-with-level S
  ∧ no-clause-is-false S
  ∧ no-smaller-confl S
  ∧ no-clause-is-false S

```

lemma *cdcl_W-stgy-cdcl_W-stgy-invariant*:

```

assumes
  cdclW: cdclW-stgy S T and
  inv-s: cdclW-stgy-invariant S and
  inv: cdclW-all-struct-inv S
shows
  cdclW-stgy-invariant T
unfolding cdclW-stgy-invariant-def cdclW-all-struct-inv-def apply standard
apply (rule cdclW-stgy-ex-lit-of-max-level[of S])
using assms unfolding cdclW-stgy-invariant-def cdclW-all-struct-inv-def apply auto[7]
apply standard
using cdclW cdclW-stgy-not-non-negated-init-clss apply blast
apply standard
apply (rule cdclW-stgy-no-smaller-confl-inv)
using assms unfolding cdclW-stgy-invariant-def cdclW-all-struct-inv-def apply auto[4]
using cdclW cdclW-stgy-not-non-negated-init-clss by auto

```

lemma *rtrancp-cdcl_W-stgy-cdcl_W-stgy-invariant*:

```

assumes

```

```

cdclW: cdclW-stgy** S T and
inv-s: cdclW-stgy-invariant S and
inv: cdclW-all-struct-inv S
shows
  cdclW-stgy-invariant T
using assms apply (induction)
  apply simp
using cdclW-stgy-cdclW-stgy-invariant rtrancp-cdclW-all-struct-inv-inv
rtrancp-cdclW-stgy-rtrancp-cdclW 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 |
cut-trail-wrt-clause C (Marked L - # M) S =
  (if -L ∈ # C then S
   else cut-trail-wrt-clause C M (decr-bt-lvl (tl-trail S))) |
cut-trail-wrt-clause C (Propagated L - # M) S =
  (if -L ∈ # C then S
   else cut-trail-wrt-clause C M (tl-trail S))

```

definition *add-new-clause-and-update* :: 'v 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.premis marked-lit.sel(1) mem-Collect-eq true-annot-def true-annot-lit-of-notin-skip
    true-annots-def true-cls-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.premis marked-lit.sel(2) mem-Collect-eq true-annot-def true-annot-lit-of-notin-skip
      true-annots-def true-cls-def zero-less-diff)
qed

lemma cut-trail-wrt-clause-hd-trail-in-or-empty-trail:
  (( $\forall L \in \#C. -L \notin \text{ lits-of } (\text{trail } T)$ )  $\wedge$  trail (cut-trail-wrt-clause C (trail T) T) = [])
   $\vee$  (-lit-of (hd (trail (cut-trail-wrt-clause C (trail T) T)))  $\in \#C$ 
     $\wedge$  length (trail (cut-trail-wrt-clause C (trail T) T))  $\geq 1$ )
  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- $cdcl_W :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** S **where**

add-conflict:

trail $S \models_{asm} \text{init-clss } S \implies \text{distinct-mset } C \implies \text{conflicting } S = C\text{-True} \implies$
 trail $S \models_{as} C\text{Not } C \implies$
 full $cdcl_W\text{-stgy}$
 (update-conflicting (C-Clause C) (add-init-cls C (cut-trail-wrt-clause C (trail S) S))) T \implies
 incremental- $cdcl_W$ S T |

add-no-conflict:

trail $S \models_{asm} \text{init-clss } S \implies \text{distinct-mset } C \implies \text{conflicting } S = C\text{-True} \implies$
 $\neg \text{trail } S \models_{as} C\text{Not } C \implies$
 full $cdcl_W\text{-stgy}$ (add-init-cls C S) T \implies
 incremental- $cdcl_W$ 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* $S \ \{\#\} \ S \mid$
add-learned-clss-plus:
add-learned-clss $S \ A \ T \implies \text{add-learned-clss } S \ (\{\#x\# \} + A) \ (\text{add-learned-clss } x \ T)$
declare *add-learned-clss.intros*[intro]

lemma *Ex-add-learned-clss*:
 $\exists T. \text{add-learned-clss } S \ A \ T$
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* $T = \text{trail } S$
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* $T = U + \text{learned-clss } S$
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* $T = \text{init-clss } S$
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* $T = \text{conflicting } S$
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* $T = \text{backtrack-lvl } S$
using *assms* **by** (*induction* *rule*: *add-learned-clss.induct*)
(*auto simp*: *ac-simps* *dest*: *add-learned-clss-trail*)

lemma *add-learned-clss-init-state-mempty*[*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 *rtranchp-cdcl_W-stgy-no-smaller-conf-inv cdcl_W-stgy-final-state-conclusive*

lemma *cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv*:

assumes
inv-T: *cdcl_W-all-struct-inv* T **and**
tr-T-N[*simp*]: *trail* $T \models_{asm} N$ **and**
tr-C[*simp*]: *trail* $T \models_{as} C \text{Not } C$ **and**

```

[simp]: distinct-mset C
shows cdclW-all-struct-inv (add-new-clause-and-update C T) (is cdclW-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  $\in$  lits-of (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 cdclW-all-struct-inv-def no-strange-atm-def add-new-clause-and-update-def
    cdclW-M-level-inv-def
    by (auto dest!: H H')

  have M-leve: cdclW-M-level-inv T
    using inv-T unfolding cdclW-all-struct-inv-def by blast
  then have no-dup (M @ trail (cut-trail-wrt-clause C (trail T) T))
    unfolding cdclW-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-leve unfolding cdclW-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]: cdclW-M-level-inv ?T
    unfolding cdclW-M-level-inv-def apply (auto dest: H H')
    simp: M-leve cdclW-M-level-inv-def cut-trail-wrt-clause-backtrack-lvl-length-marked)
    using M-leve cut-trail-wrt-clause-get-all-levels-of-marked[of T C]
    by (auto simp: cdclW-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  $\models_{as} C \text{Not } C$ 
    by (simp add: cut-trail-wrt-clause-CNot-trail)
  then have [simp]: cdclW-conflicting ?T
    unfolding cdclW-conflicting-def apply simp
    by (metis M <cdclW-conflicting T> 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) ∈ 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) ∈ set (get-all-marked-decomposition (trail T))
    using M by simp metis
  then have ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘set a ∪ set-mset (init-clss ?T)
     $\models_{ps}$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘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 ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘set a ∪ set-mset (init-clss ?T)
     $\models_{ps}$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘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 all-decomposition-implies-m (init-clss ?T)
  (get-all-marked-decomposition (trail ?T))
  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]: trail T  $\models_{as}$  CNot C and
    [simp]: distinct-mset C
  shows cdclW-stgy-invariant (add-new-clause-and-update C T) (is cdclW-stgy-invariant ?T')
proof –
  have cdclW-all-struct-inv ?T'
    using cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv assms by blast
  then have
    no-dup-cut-T[simp]: no-dup (trail (cut-trail-wrt-clause C (trail T) T)) and
    n-d[simp]: no-dup (trail T)
    using cdclW-M-level-inv-decomp(2) cdclW-all-struct-inv-def inv
    n-dup-no-dup-trail-cut-trail-wrt-clause by blast+
  then have trail (add-new-clause-and-update C T)  $\models_{as}$  CNot C
    by (simp add: add-new-clause-and-update-def cut-trail-wrt-clause-CNot-trail
      cdclW-M-level-inv-def cdclW-all-struct-inv-def)
  obtain MT where
    MT: trail T = MT @ trail (cut-trail-wrt-clause C (trail T) T)

```

```

using trail-cut-trail-wrt-clause by blast
consider
  (false)  $\forall L \in \#C. - L \notin \text{ lits-of } (\text{trail } T)$  and trail (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ ) = []
  | (not-false)  $- \text{ lit-of } (\text{hd } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))) \in \# C$  and
     $1 \leq \text{length } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))$ 
using cut-trail-wrt-clause-hd-trail-in-or-empty-trail[ $\text{of } C T$ ] by auto
then show ?thesis
proof cases
  case false note  $C = \text{this}(1)$  and  $\text{empty-tr} = \text{this}(2)$ 
  then have [simp]:  $C = \{\#\}$ 
    by (simp add: in-CNot-implies-uminus(2) multiset-eqI)
  show ?thesis
    using empty-tr unfolding cdclW-stgy-invariant-def no-smaller-confl-def
    cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
next
  case not-false note  $C = \text{this}(1)$  and  $l = \text{this}(2)$ 
  let ?L =  $- \text{ lit-of } (\text{hd } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))$ 
  have get-all-levels-of-marked (trail (add-new-clause-and-update  $C T$ )) =
    rev [1.. $1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } (\text{add-new-clause-and-update } C T)))$ ]
    using <cdclW-all-struct-inv ?T'> unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
    by blast
  moreover
    have backtrack-lvl (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ ) =
      length (get-all-levels-of-marked (trail (add-new-clause-and-update  $C T$ )))
      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)
  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)
    then have atm-of ?L  $\notin \text{ atm-of ' lits-of } (\text{tl } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))$ 
      apply (cases trail (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ ))
      apply (auto)
      using Marked-Propagated-in-iff-in-lits-of defined-lit-map by blast
  ultimately have  $L: \text{get-level } (- ?L) (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))$ 
    = length (get-all-levels-of-marked (trail (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ )))
    using get-level-get-rev-level-get-all-levels-of-marked[OF
      <atm-of ?L  $\notin \text{ atm-of ' lits-of } (\text{tl } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))$ >,
      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  $L': \text{length } (\text{get-all-levels-of-marked } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))$ 
    = 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)  $D \in \# \text{ clauses } T$ 
    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  $-\text{[simp]} = \text{this}$ 
  then have atm-of ( $-\text{?L}$ )  $\in \text{atm-of ' (lits-of } M)$ 
    using 1(3) C in-CNot-implies-uminus(2) by blast
  moreover
    have lit-of (hd (M' @ Marked K i # []))  $= -\text{?L}$ 
      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 ( $-\text{?L}$ )  $\in \text{atm-of ' (lits-of (M' @ Marked K i # []))}$ 
      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 cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
qed
qed

```

lemma *full-cdcl_W-stgy-inv-normal-form:*

assumes

full: full cdcl_W-stgy S T **and**

inv-s: cdcl_W-stgy-invariant S **and**

inv: cdcl_W-all-struct-inv S

shows *conflicting T = C-Clause {#} \wedge unsatisfiable (set-mset (init-cls S))*

\vee *conflicting T = C-True \wedge trail T \models_{asm} init-cls S \wedge satisfiable (set-mset (init-cls S))*

proof –

have *no-step cdcl_W-stgy T*

using *full unfolding full-def* **by** *blast*

moreover have *cdcl_W-all-struct-inv T* **and** *inv-s: cdcl_W-stgy-invariant T*

apply (*metis cdcl_W-ops.rtranclp-cdcl_W-stgy-rtranclp-cdcl_W cdcl_W-ops-axioms full full-def inv*)

```

    rtrancpl-cdclW-all-struct-inv-inv)
  by (metis full full-def inv inv-s rtrancpl-cdclW-stgy-cdclW-stgy-invariant)
ultimately have conflicting  $T = C\text{-Clause } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } T))$ 
   $\vee$  conflicting  $T = C\text{-True} \wedge \text{trail } T \models_{\text{asm}} \text{init-clss } T$ 
  using cdclW-stgy-final-state-conclusive[of  $T$ ] full
  unfolding cdclW-all-struct-inv-def cdclW-stgy-invariant-def full-def by fast
moreover have consistent-interp (lits-of (trail  $T$ ))
  using  $\langle \text{cdcl}_W\text{-all-struct-inv } T \rangle$  unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
  by auto
moreover have init-clss  $S = \text{init-clss } T$ 
  using inv unfolding cdclW-all-struct-inv-def
  by (metis rtrancpl-cdclW-stgy-no-more-init-clss full full-def)
ultimately show ?thesis
  by (metis satisfiable-carac' true-annot-def true-annots-def true-clss-def)
qed

```

lemma incremental-cdcl_W-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
proof (induction)
case (add-conf  $C$   $T$ )
let ? $T = (\text{update-conflicting } (C\text{-Clause } C) (\text{add-init-clss } C (\text{cut-trail-wrt-clause } C (\text{trail } S) S)))$ 
have cdclW-all-struct-inv ? $T$  and inv-s- $T$ : cdclW-stgy-invariant ? $T$ 
  using add-conf.hyps(1,2,4) add-new-clause-and-update-def
  cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv inv apply auto[1]
  using add-conf.hyps(1,2,4) add-new-clause-and-update-def
  cdclW-all-struct-inv-add-new-clause-and-update-cdclW-stgy-inv inv s-inv by auto
case 1 show ?case
  by (metis add-conf.hyps(1,2,4,5) add-new-clause-and-update-def
    cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv
    rtrancpl-cdclW-all-struct-inv-inv rtrancpl-cdclW-stgy-rtrancpl-cdclW full-def inv)

case 2 show ?case
  by (metis inv-s- $T$  add-conf.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
    rtrancpl-cdclW-stgy-cdclW-stgy-invariant)
next
case (add-no-conf  $C$   $T$ )
case 1
have cdclW-all-struct-inv (add-init-clss  $C$   $S$ )
  using inv  $\langle \text{distinct-mset } C \rangle$  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-conf(5) unfolding full-def by (auto intro: rtrancpl-cdclW-stgy-cdclW-all-struct-inv)
case 2 have cdclW-stgy-invariant (add-init-clss  $C$   $S$ )
  using s-inv  $\langle \neg \text{trail } S \models_{\text{as}} C\text{Not } C \rangle$  inv unfolding cdclW-stgy-invariant-def no-smaller-conf-def
  eq-commute[of - trail -] cdclW-M-level-inv-def cdclW-all-struct-inv-def
  by (auto simp: true-annots-true-clss-def iff-negation-in-model clauses-def split: split-if-asm)

```


then show *?case*
by (*metis* $\langle \text{cdcl}_W\text{-all-struct-inv } (\text{add-init-cls } C \ S) \rangle \text{ add-no-confl.hyps}(5) \text{ full-def}$
 $\text{rtrancpl-cdcl}_W\text{-stgy-cdcl}_W\text{-stgy-invariant}$)
qed

lemma *rtrancpl-incremental-cdcl_W-inv*:

assumes
inc: *incremental-cdcl_W** S T* **and**
inv: *cdcl_W-all-struct-inv S* **and**
s-inv: *cdcl_W-stgy-invariant S*
shows
cdcl_W-all-struct-inv T **and**
cdcl_W-stgy-invariant T
using *inc* **apply** *induction*
using *inv* **apply** *simp*
using *s-inv* **apply** *simp*
using *incremental-cdcl_W-inv* **by** *blast+*

lemma *incremental-conclusive-state*:

assumes
inc: *incremental-cdcl_W S T* **and**
inv: *cdcl_W-all-struct-inv S* **and**
s-inv: *cdcl_W-stgy-invariant S*
shows *conflicting T = C-Clause {#} \wedge unsatisfiable (set-mset (init-clss T))*
 \vee *conflicting T = C-True \wedge trail T \models_{asm} init-clss T \wedge satisfiable (set-mset (init-clss T))*
using *inc* **apply** *induction*

apply (*metis* *Nitpick.rtrancpl-unfold add-confl full-cdcl_W-stgy-inv-normal-form full-def*
incremental-cdcl_W-inv(1) incremental-cdcl_W-inv(2) inv s-inv)
by (*metis* (*full-types*) *rtrancpl-unfold add-no-confl full-cdcl_W-stgy-inv-normal-form*
full-def incremental-cdcl_W-inv(1) incremental-cdcl_W-inv(2) inv s-inv)

lemma *trancpl-incremental-correct*:

assumes
inc: *incremental-cdcl_W⁺⁺ S T* **and**
inv: *cdcl_W-all-struct-inv S* **and**
s-inv: *cdcl_W-stgy-invariant S*
shows *conflicting T = C-Clause {#} \wedge unsatisfiable (set-mset (init-clss T))*
 \vee *conflicting T = C-True \wedge trail T \models_{asm} init-clss T \wedge satisfiable (set-mset (init-clss T))*
using *inc* **apply** *induction*
using *assms incremental-conclusive-state* **apply** *blast*
by (*meson incremental-conclusive-state inv rtrancpl-incremental-cdcl_W-inv s-inv*
trancpl-into-rtrancpl)

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** $n\text{-d} = \text{this}(2)$ **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 $n\text{-d}$ **by** *auto*
next
case (*Suc n*) **note** $IH = \text{this}(1)$ **and** $n = \text{this}(2)$ **and** $n\text{-d} = \text{this}(3)$ **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' $n\text{-d}$ **unfolding** M **by** *auto*
moreover have $L' \notin \{L' \in \text{set } M''. \text{is-marked } L'\}$
using $n\text{-d}$ **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' $n\text{-d}$ M **by** *auto*
then show ?*case* **using** *append*[of $L' \# M''$ L M] nm L $n\text{-d}$ **unfolding** M **by** *blast*
qed

lemma *trail-bloc-induction:*

assumes
 $n\text{-d}: \text{no-dup } M$ **and**
 $nil: P []$ **and**
 $\text{append}: \bigwedge M L M'. P M \implies \text{is-marked } L \implies \forall m \in \text{set } M'. \neg \text{is-marked } m \implies \text{no-dup } (L \# M' @ M) \implies$
 $P (L \# M' @ M)$ **and**
 $\text{append-nm}: \bigwedge M' M''. P M' \implies M = M'' @ M' \implies \forall m \in \text{set } M''. \neg \text{is-marked } m \implies P M$
shows
 $P M$
proof (*cases* $\{L' \in \text{set } M. \text{is-marked } L'\} = \{\}$)
case *True*
then show ?*thesis* **using** *append-nm*[of [] M] nil **by** *auto*
next
case *False*
then have $\exists L' \in \text{set } M. \text{is-marked } L'$
by *auto*
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 $P (L' \# M'')$
apply (*rule blocked-induction-with-marked*)
using $n\text{-d}$ **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  $\Rightarrow$  ('v, nat, 'v clause) marked-lits  $\Rightarrow$  bool
for M :: ('v, nat, 'v clause) marked-lits where
  Tcons M [] |
  Tcons M M'  $\Rightarrow$  M = M'' @ M'  $\Rightarrow$  ( $\forall m \in \text{set } M''. \neg \text{is-marked } m$ )  $\Rightarrow$  Tcons M (M'' @ M') |
  Tcons M M'  $\Rightarrow$  is-marked L  $\Rightarrow$  M = M''' @ L # M'' @ M'  $\Rightarrow$  ( $\forall m \in \text{set } M''. \neg \text{is-marked } m$ )  $\Rightarrow$ 
  Tcons M (L # M'' @ M')

```

```

lemma Tcons-same-end: Tcons M M'  $\Rightarrow$   $\exists M''. M = M'' @ M'$ 
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 that appear in the state are not related to the 2-watched algorithm.

```

datatype 'v twl-clause =
  TWL-Clause (watched: 'v clause) (unwatched: 'v clause)

```

```

abbreviation raw-clause :: 'v twl-clause  $\Rightarrow$  'v clause where
  raw-clause C  $\equiv$  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  $\equiv$  image-mset raw-clause (init-clss S)

```

```

abbreviation raw-learned-clss where
  raw-learned-clss S  $\equiv$  image-mset raw-clause (learned-clss S)

```

```

abbreviation clauses where
  clauses S  $\equiv$  init-clss S + learned-clss S

```

```

abbreviation raw-clauses where
  raw-clauses S  $\equiv$  image-mset raw-clause (clauses S)

```

```

definition
  candidates-propagate :: ('v, 'vl, 'mark) twl-state  $\Rightarrow$  ('v literal  $\times$  'v clause) set
where
  candidates-propagate S =

```

$\{(L, \text{raw-clause } C) \mid L \ C.$
 $C \in \# \text{ clauses } S \wedge \text{watched } C - \text{mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S)) = \{\#L\# \} \wedge$
 $\text{undefined-lit } (\text{trail } S) \ L\}$

definition *candidates-conflict* :: ('v, 'wl, 'mark) twl-state \Rightarrow 'v 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 list \Rightarrow 'a \Rightarrow 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 about updates: if there is a literal L with $-L$ in the trail, and L is not watched, then it stays unwatched; i.e., while updating with *rewatch* it does not get swap with a watched literal L' such that $-L'$ is in the trail.

primrec *watched-decided-most-recently* :: ('v, 'wl, 'mark) marked-lit list \Rightarrow 'v twl-clause \Rightarrow bool
where
watched-decided-most-recently $M \ (\text{TWL-Clause } W \ UW) \longleftrightarrow$
 $(\forall L' \in \# W. \forall L \in \# UW. \\$
 $-L' \in \text{lits-of } M \longrightarrow -L \in \text{lits-of } M \longrightarrow L \notin \# W \longrightarrow \\$
 $\text{index } (\text{map lit-of } M) \ (-L') \leq \text{index } (\text{map lit-of } M) \ (-L))$

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-tw-cl* :: ('v, 'wl, 'mark) marked-lit list \Rightarrow 'v twl-clause \Rightarrow bool **where**
wf-tw-cl $M \ (\text{TWL-Clause } W \ UW) \longleftrightarrow$
 $\text{distinct-mset } W \wedge \text{size } W \leq 2 \wedge (\text{size } W < 2 \longrightarrow \text{set-mset } UW \subseteq \text{set-mset } W) \wedge$
 $(\forall L \in \# W. -L \in \text{lits-of } M \longrightarrow (\forall L' \in \# UW. L' \notin \# W \longrightarrow -L' \in \text{lits-of } M)) \wedge$
watched-decided-most-recently $M \ (\text{TWL-Clause } W \ UW)$

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*

lemma *wf-tw-cl-annotation-indepndant*:
assumes $M: \text{map lit-of } M = \text{map lit-of } M'$
shows $\text{wf-tw-cl } M \ (\text{TWL-Clause } W \ UW) \longleftrightarrow \text{wf-tw-cl } M' \ (\text{TWL-Clause } W \ UW)$

proof –
have $\text{lits-of } M = \text{lits-of } M'$
using *arg-cong* [*OF* M , *of set*] **by** (*simp add: lits-of-def*)
then show *?thesis*
by (*simp add: lits-of-def* M)
qed

lemma *wf-tw-cl-wf-tw-cl-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 ⟨count W L' = 0⟩
      by (metis (no-types, lifting) C M bspec-mset insert-iff less-not-refl2 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
    assume
      L' ∈# W and
      L ∈# UW and
      L'M: − L' ∈ lits-of M' and
      − L ∈ lits-of M' and
      L ∉# W
    moreover
      have lit-of l ≠ − L'
      using n-d unfolding M
      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') (− L)
      by (fastforce 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 (auto simp add: M C)
qed

```

definition *wf-twl-state* :: ('*v*, '*wl*, '*mark*) *twl-state* ⇒ *bool* **where**

$wf\text{-}twl\text{-}state\ S \longleftrightarrow (\forall C \in \# \text{ clauses } S. \text{ wf-}twl\text{-}cls\ (trail\ S)\ C) \wedge no\text{-}dup\ (trail\ S)$

lemma *wf-candidates-propagate-sound*:

assumes *wf*: *wf-twl-state S* **and**

cand: $(L, C) \in \text{candidates-propagate } S$

shows $trail\ S \models_{as} C \text{Not } (mset\text{-}set\ (set\text{-}mset\ C - \{L\})) \wedge \text{undefined-lit } (trail\ S)\ L$

proof

def $M \equiv trail\ S$

def $N \equiv \text{init-clss } S$

def $U \equiv \text{learned-clss } S$

note $MNU\text{-}defs\ [simp] = M\text{-}def\ N\text{-}def\ U\text{-}def$

obtain Cw **where** *cw*:

$C = \text{raw-clause } Cw$

$Cw \in \# N + U$

$\text{watched } Cw - mset\text{-}set\ (\text{uminus } ' \text{ lits-of } M) = \{\#L\#\}$

$\text{undefined-lit } M\ L$

using *cand* **unfolding** *candidates-propagate-def MNU-defs* **by** *blast*

obtain $W\ UW$ **where** *cw-eq*: $Cw = \text{TWL-Clause } W\ UW$

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 twl-clause.sel(1)*)

have *wf-c*: *wf-twl-cls M Cw*

using *wf* ($Cw \in \# N + U$) **unfolding** *wf-twl-state-def* **by** *simp*

have *w-nw*:

distinct-mset W

$\text{size } W < 2 \implies \text{set-mset } UW \subseteq \text{set-mset } W$

$\bigwedge L\ L'. L \in \# W \implies -L \in \text{lits-of } M \implies L' \in \# UW \implies L' \notin \# W \implies -L' \in \text{lits-of } M$

using *wf-c* **unfolding** *cw-eq* **by** *auto*

have $\forall L' \in \text{set-mset } C - \{L\}. -L' \in \text{lits-of } M$

proof (*cases size W < 2*)

case *True*

moreover **have** *size W ≠ 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 twl-clause.sel(1))

from *True* **have** $\text{set-mset } UW \subseteq \text{set-mset } W$

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' : $L' \in \text{set-mset } C - \{L\}$

```

have ex-la:  $\exists La. La \neq L \wedge La \in \# W$ 
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:  $La \neq L \wedge La \in \# W$ 
  by blast
then have  $La \in \# \text{mset-set } (\text{uminus } ' \text{ lits-of } M)$ 
  using cw(3)[unfolded cw-eq, simplified, folded M-def]
  by (metis count-diff count-single diff-zero not-gr0)
then have nla:  $-La \in \text{lits-of } M$ 
  by auto
then show  $-L' \in \text{lits-of } M$ 

proof -
  have f1:  $L' \in \text{set-mset } C$ 
    using l' by blast
  have f2:  $L' \notin \{L\}$ 
    using l' by fastforce
  have  $\bigwedge l L. - (l::'a \text{ literal}) \in L \vee l \notin \text{uminus } ' L$ 
    by force
  then have  $\bigwedge l. - l \in \text{lits-of } M \vee \text{count } \{\#L\} l = \text{count } (C - UW) l$ 
    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  $\text{trail } S \models_{\text{as}} C \text{Not } (\text{mset-set } (\text{set-mset } C - \{L\}))$ 
  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:  $C \in \# \text{raw-clauses } S$  and

```

```

  l-mem:  $L \in \# C$  and
  unsat:  $\text{trail } S \models_{\text{as}} \text{CNot } (\text{mset-set } (\text{set-mset } C - \{L\}))$  and
  undef:  $\text{undefined-lit } (\text{trail } S) L$ 
shows  $(L, C) \in \text{candidates-propagate } S$ 
proof -
  def  $M \equiv \text{trail } S$ 
  def  $N \equiv \text{init-clss } S$ 
  def  $U \equiv \text{learned-clss } S$ 

  note  $MNU\text{-defs } [\text{simp}] = M\text{-def } N\text{-def } U\text{-def}$ 

  obtain  $Cw$  where  $cw: C = \text{raw-clause } Cw$   $Cw \in \# N + U$ 
  using  $c\text{-mem}$  by force

  obtain  $W UW$  where  $cw\text{-eq}: Cw = \text{TWL-Clause } W UW$ 
  by ( $\text{case-tac } Cw, \text{blast}$ )

  have  $wf\text{-c}: wf\text{-twl-clss } M Cw$ 
  using  $wf\ cw(2)$  unfolding  $wf\text{-twl-state-def}$  by simp

  have  $w\text{-nw}$ :
    distinct-mset  $W$ 
    size  $W < 2 \implies \text{set-mset } UW \subseteq \text{set-mset } W$ 
     $\bigwedge L L'. L \in \# W \implies -L \in \text{lits-of } M \implies L' \in \# UW \implies L' \notin \# W \implies -L' \in \text{lits-of } M$ 
  using  $wf\text{-c}$  unfolding  $cw\text{-eq}$  by auto

  have  $\text{unit-set}: \text{set-mset } (W - \text{mset-set } (\text{uminus } ' \text{lits-of } M)) = \{L\}$ 
  proof
    show  $\text{set-mset } (W - \text{mset-set } (\text{uminus } ' \text{lits-of } M)) \subseteq \{L\}$ 
    proof
      fix  $L'$ 
      assume  $l': L' \in \text{set-mset } (W - \text{mset-set } (\text{uminus } ' \text{lits-of } M))$ 
      hence  $l'\text{-mem-w}: L' \in \text{set-mset } W$ 
      by auto
      have  $L' \notin \text{uminus } ' \text{lits-of } M$ 
      using  $\text{distinct-mem-diff-mset}[OF\ w\text{-nw}(1)\ l']$  by simp
      then have  $\neg M \models_a \{\# - L'\#\}$ 
      using  $\text{image-iff}$  by fastforce
      moreover have  $L' \in \# C$ 
      using  $cw(1)\ cw\text{-eq}\ l'\text{-mem-w}$  by auto
      ultimately have  $L' = L$ 
      unfolding  $M\text{-def}$  by ( $\text{metis unsat}[\text{unfolded } \text{CNot-def true-annots-def, simplified}]$ )
      then show  $L' \in \{L\}$ 
      by simp
    qed
  qed
next
  show  $\{L\} \subseteq \text{set-mset } (W - \text{mset-set } (\text{uminus } ' \text{lits-of } M))$ 
  proof clarify
    have  $L \in \# W$ 
    proof (cases  $W$ )
      case empty
      thus ?thesis
      using  $w\text{-nw}(2)\ cw(1)\ cw\text{-eq}\ l\text{-mem}$  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  $\text{ trail } S \models_{\text{as}} \text{ CNot } C \wedge C \in \# \text{ image-mset raw-clause } (\text{ clauses } S)$ 
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-twl-state S and
    c-mem: C ∈# raw-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 ($case\text{-tac } Cw, \text{blast}$)

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 $w\text{-nw}$:
 $distinct\text{-mset } W$
 $size \text{ } W < 2 \implies set\text{-mset } UW \subseteq set\text{-mset } W$
 $\bigwedge L \text{ } L'. L \in \# \text{ } W \implies -L \in \text{lits-of } M \implies L' \in \# \text{ } UW \implies L' \notin \# \text{ } W \implies -L' \in \text{lits-of } M$
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** $unsat[unfolding \text{ } CNot\text{-def } true\text{-annots-def}, simplified]$ **by** $blast$
then have $set\text{-mset } C \subseteq uminus \text{ ' } \text{lits-of } M$
by ($metis \text{ imageI mem-set-mset-iff subsetI uminus-of-uminus-id}$)
then have $set\text{-mset } W \subseteq uminus \text{ ' } \text{lits-of } M$
using $cw(1) \text{ } cw\text{-eq}$ **by** $auto$
then have $subset: W \subseteq \# \text{ } mset\text{-set } (uminus \text{ ' } \text{lits-of } M)$
by ($simp \text{ add: } w\text{-nw}(1)$)

have $W = \text{watched } Cw$
using $cw\text{-eq } twl\text{-clause.sel}(1)$ **by** $simp$
then show $?thesis$
using $MNU\text{-defs } cw(1) \text{ } cw(2) \text{ } subset \text{ } candidates\text{-conflict-def}$ **by** $blast$
qed

typedef $'v \text{ } wf\text{-twl} = \{S :: ('v, nat, 'v \text{ clause}) \text{ twl-state. } wf\text{-twl-state } S\}$
morphisms $rough\text{-state-of-twl } twl\text{-of-rough-state}$
proof –
have $TWL\text{-State } ([::('v, nat, 'v \text{ clause}) \text{ marked-lits})$
 $\{\#\} \{\#\} 0 \text{ } C\text{-True} \in \{S :: ('v, nat, 'v \text{ clause}) \text{ twl-state. } wf\text{-twl-state } S\}$
by ($auto \text{ simp: } wf\text{-twl-state-def}$)
then show $?thesis$ **by** $auto$
qed

lemma $wf\text{-twl-state-rough-state-of-twl}[simp]: wf\text{-twl-state } (rough\text{-state-of-twl } S)$
using $rough\text{-state-of-twl}$ **by** $auto$

abbreviation $candidates\text{-conflict-twl} :: 'v \text{ } wf\text{-twl} \Rightarrow 'v \text{ literal multiset set}$ **where**
 $candidates\text{-conflict-twl } S \equiv candidates\text{-conflict } (rough\text{-state-of-twl } S)$

abbreviation $candidates\text{-propagate-twl} :: 'v \text{ } wf\text{-twl} \Rightarrow ('v \text{ literal} \times 'v \text{ clause}) \text{ set}$ **where**
 $candidates\text{-propagate-twl } S \equiv candidates\text{-propagate } (rough\text{-state-of-twl } S)$

abbreviation $trail\text{-twl} :: 'a \text{ } wf\text{-twl} \Rightarrow ('a, nat, 'a \text{ literal multiset}) \text{ marked-lit list}$ **where**
 $trail\text{-twl } S \equiv trail \text{ } (rough\text{-state-of-twl } S)$

abbreviation $clauses\text{-twl} :: 'a \text{ } wf\text{-twl} \Rightarrow 'a \text{ literal multiset multiset}$ **where**
 $clauses\text{-twl } S \equiv raw\text{-clauses } (rough\text{-state-of-twl } S)$

abbreviation $init\text{-clss-twl} :: 'a \text{ } wf\text{-twl} \Rightarrow 'a \text{ literal multiset multiset}$ **where**
 $init\text{-clss-twl } S \equiv raw\text{-init-clss } (rough\text{-state-of-twl } S)$

abbreviation $learned\text{-clss-twl} :: 'a \text{ } wf\text{-twl} \Rightarrow 'a \text{ literal multiset multiset}$ **where**

learned-clss-twl $S \equiv \text{raw-learned-clss } (\text{rough-state-of-twl } S)$

abbreviation *backtrack-lvl-twl* **where**

backtrack-lvl-twl $S \equiv \text{backtrack-lvl } (\text{rough-state-of-twl } S)$

abbreviation *conflicting-twl* **where**

conflicting-twl $S \equiv \text{conflicting } (\text{rough-state-of-twl } S)$

lemma *wf-candidates-twl-conflict-complete*:

assumes

c-mem: $C \in \# \text{ clauses-twl } S$ **and**

unsat: $\text{trail-twl } S \models_{\text{as}} C \text{Not } C$

shows $C \in \text{candidates-conflict-twl } S$

using *c-mem unsat wf-candidates-conflict-complete wf-twl-state-rough-state-of-twl* **by** *blast*

locale *abstract-twl* =

fixes

watch :: $('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ twl-clause}$ **and**

rewatch :: $('v, \text{nat}, 'v \text{ literal multiset}) \text{ marked-lit} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$

$'v \text{ twl-clause} \Rightarrow 'v \text{ twl-clause}$ **and**

linearize :: $'v \text{ clauses} \Rightarrow 'v \text{ clause list}$ **and**

restart-learned :: $('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow 'v \text{ twl-clause multiset}$

assumes

clause-watch: $\text{no-dup}(\text{trail } S) \implies \text{raw-clause } (\text{watch } S C) = C$ **and**

wf-watch: $\text{no-dup } (\text{trail } S) \implies \text{wf-twl-cls } (\text{trail } S) (\text{watch } S C)$ **and**

clause-rewatch: $\text{raw-clause } (\text{rewatch } L S C') = \text{raw-clause } C'$ **and**

wf-rewatch:

$\text{no-dup } (\text{trail } S) \implies \text{undefined-lit } (\text{trail } S) (\text{lit-of } L) \implies \text{wf-twl-cls } (\text{trail } S) C' \implies$

$\text{wf-twl-cls } (L \# \text{ trail } S) (\text{rewatch } L S C')$

and

linearize: $\text{mset } (\text{linearize } N) = N$ **and**

restart-learned: $\text{restart-learned } S \subseteq \# \text{ learned-clss } S$

begin

lemma *linearize-mempty[simp]*: $\text{linearize } \{\#\} = []$

using *linearize mset-zero-iff* **by** *blast*

definition

cons-trail :: $('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$

$('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$

where

cons-trail $L S =$

$\text{TWL-State } (L \# \text{ trail } S) (\text{image-mset } (\text{rewatch } L S) (\text{init-clss } S))$

$(\text{image-mset } (\text{rewatch } L S) (\text{learned-clss } S)) (\text{backtrack-lvl } S) (\text{conflicting } S)$

definition

add-init-cls :: $'v \text{ clause} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$

$('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$

where

add-init-cls $C S =$

$\text{TWL-State } (\text{trail } S) (\{\# \text{ watch } S C \# \} + \text{init-clss } S) (\text{learned-clss } S) (\text{backtrack-lvl } S)$

$(\text{conflicting } S)$

definition

add-learned-cls :: $'v \text{ clause} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$

$(v, nat, v \text{ clause}) \text{ twl-state}$
where
 $add\text{-}learned\text{-}cls \ C \ S =$
 $TWL\text{-}State \ (trail \ S) \ (init\text{-}clss \ S) \ (\{\#watch \ S \ C\# \} + learned\text{-}clss \ S) \ (backtrack\text{-}lvl \ S)$
 $(conflicting \ S)$

definition
 $remove\text{-}cls :: v \text{ clause} \Rightarrow (v, nat, v \text{ clause}) \text{ twl-state} \Rightarrow (v, nat, v \text{ clause}) \text{ twl-state}$
where
 $remove\text{-}cls \ C \ S =$
 $TWL\text{-}State \ (trail \ S) \ (filter\text{-}mset \ (\lambda D. \text{raw-clause } D \neq C) \ (init\text{-}clss \ S))$
 $(filter\text{-}mset \ (\lambda D. \text{raw-clause } D \neq C) \ (learned\text{-}clss \ S)) \ (backtrack\text{-}lvl \ S)$
 $(conflicting \ S)$

definition $init\text{-}state :: v \text{ clauses} \Rightarrow (v, nat, v \text{ clause}) \text{ twl-state}$ **where**
 $init\text{-}state \ N = fold \ add\text{-}init\text{-}cls \ (linearize \ N) \ (TWL\text{-}State \ [] \ \{\#\} \ \{\#\} \ 0 \ C\text{-}True)$

lemma $unchanged\text{-}fold\text{-}add\text{-}init\text{-}cls$:
 $trail \ (fold \ add\text{-}init\text{-}cls \ Cs \ (TWL\text{-}State \ M \ N \ U \ k \ C)) = M$
 $learned\text{-}clss \ (fold \ add\text{-}init\text{-}cls \ Cs \ (TWL\text{-}State \ M \ N \ U \ k \ C)) = U$
 $backtrack\text{-}lvl \ (fold \ add\text{-}init\text{-}cls \ Cs \ (TWL\text{-}State \ M \ N \ U \ k \ C)) = k$
 $conflicting \ (fold \ add\text{-}init\text{-}cls \ Cs \ (TWL\text{-}State \ M \ N \ U \ k \ C)) = C$
by $(induct \ Cs \text{ arbitrary: } N) \ (auto \ simp: add\text{-}init\text{-}cls\text{-}def)$

lemma $unchanged\text{-}init\text{-}state[simp]$:
 $trail \ (init\text{-}state \ N) = []$
 $learned\text{-}clss \ (init\text{-}state \ N) = \{\#\}$
 $backtrack\text{-}lvl \ (init\text{-}state \ N) = 0$
 $conflicting \ (init\text{-}state \ N) = C\text{-}True$
unfolding $init\text{-}state\text{-}def$ **by** $(rule \ unchanged\text{-}fold\text{-}add\text{-}init\text{-}cls)+$

lemma $clauses\text{-}init\text{-}fold\text{-}add\text{-}init$:
 $no\text{-}dup \ M \Longrightarrow$
 $image\text{-}mset \ \text{raw-clause} \ (init\text{-}clss \ (fold \ add\text{-}init\text{-}cls \ Cs \ (TWL\text{-}State \ M \ N \ U \ k \ C))) =$
 $mset \ Cs + image\text{-}mset \ \text{raw-clause} \ N$
by $(induct \ Cs \text{ arbitrary: } N) \ (auto \ simp: add.assoc \ add\text{-}init\text{-}cls\text{-}def \ clause\text{-}watch)$

lemma $init\text{-}clss\text{-}init\text{-}state[simp]$: $image\text{-}mset \ \text{raw-clause} \ (init\text{-}clss \ (init\text{-}state \ N)) = N$
unfolding $init\text{-}state\text{-}def$ **by** $(simp \ add: clauses\text{-}init\text{-}fold\text{-}add\text{-}init \ linearize)$

definition $update\text{-}backtrack\text{-}lvl$ **where**
 $update\text{-}backtrack\text{-}lvl \ k \ S =$
 $TWL\text{-}State \ (trail \ S) \ (init\text{-}clss \ S) \ (learned\text{-}clss \ S) \ k \ (conflicting \ S)$

definition $update\text{-}conflicting$ **where**
 $update\text{-}conflicting \ C \ S = TWL\text{-}State \ (trail \ S) \ (init\text{-}clss \ S) \ (learned\text{-}clss \ S) \ (backtrack\text{-}lvl \ S) \ C$

definition $tl\text{-}trail$ **where**
 $tl\text{-}trail \ S =$
 $TWL\text{-}State \ (tl \ (trail \ S)) \ (init\text{-}clss \ S) \ (learned\text{-}clss \ S) \ (backtrack\text{-}lvl \ S) \ (conflicting \ S)$

definition $restart'$ **where**
 $restart' \ S = TWL\text{-}State \ [] \ (init\text{-}clss \ S) \ (restart\text{-}learned \ S) \ 0 \ C\text{-}True$
end

definition *pull* :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list **where**
pull *p xs* = *filter* *p xs* @ *filter* (*Not* o *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*))

thm *rev-cases*

lemma *list-cases2*:
fixes *l* :: 'a list
assumes
 l = [] \Longrightarrow *P* **and**
 $\bigwedge x. l = [x] \Longrightarrow P$ **and**
 $\bigwedge x y xs. l = x \# y \# xs \Longrightarrow P$
shows *P*
by (*metis* *assms list.collapse*)

lemma *filter-in-list-prop-verifiedD*:
assumes [*L* \leftarrow *P* . *Q* *L*] = *l*
shows $\forall x \in \text{set } l. x \in \text{set } P \wedge Q x$
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 *watch-nat-lists-disjointD*:
assumes
 l: [*L* \leftarrow *remdups* (*sorted-list-of-set* (*set-mset* *C*)) . - *L* $\notin \text{ lits-of } (\text{trail } S)$] = *l* **and**
 l': [*L* \leftarrow *map* ($\lambda L. - \text{lit-of } L$) (*trail* *S*) . *L* $\in \# C$] = *l'*
shows $\forall x \in \text{set } l. \forall y \in \text{set } l'. x \neq y$
by (*auto simp: l[symmetric] l'[symmetric] lits-of-def*)

lemma *watch-nat-list-cases* [*consumes* 1, *case-names* *nil-nil nil-single nil-other single-nil*
single-other other]:

fixes $C :: 'v::\text{linorder literal multiset}$ **and** $S :: ('v, 'a, 'b) \text{ twl-state}$
defines
 $xs \equiv [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \text{ . } - L \notin \text{lits-of } (\text{trail } S)]$ **and**
 $ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) \text{ . } L \in \# C]$
assumes $n\text{-d}$: $\text{no-dup } (\text{trail } S)$ **and**
 nil-nil : $xs = [] \implies ys = [] \implies P$ **and**
 nil-single :
 $\bigwedge a. xs = [] \implies ys = [a] \implies a \in \# C \implies P$ **and**
 nil-other : $\bigwedge a \ b \ ys'. xs = [] \implies ys = a \# b \# ys' \implies a \neq b \implies P$ **and**
 single-nil : $\bigwedge a. xs = [a] \implies ys = [] \implies P$ **and**
 single-other : $\bigwedge a \ b \ ys'. xs = [a] \implies ys = b \# ys' \implies a \neq b \implies P$ **and**
 other : $\bigwedge a \ b \ xs'. xs = a \# b \# xs' \implies a \neq b \implies P$
shows P
proof –
note $xs\text{-def}[simp]$ **and** $ys\text{-def}[simp]$
have dist : $\text{distinct } [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \text{ . } - L \notin \text{lits-of } (\text{trail } S)]$
by auto
then have H : $\bigwedge a \ xs. [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \text{ . } - L \notin \text{lits-of } (\text{trail } S)]$
 $\neq a \# a \# xs$
by force
show $?thesis$
apply ($\text{cases } [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \text{ . } - L \notin \text{lits-of } (\text{trail } S)]$
 rule: list-cases2 ;
 $\text{cases } [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) \text{ . } L \in \# C]$ rule: list-cases2)
using nil-nil **apply** simp
using nil-single **apply** ($\text{force dest: filter-in-list-prop-verifiedD}$)
using nil-other
apply ($\text{auto dest: filter-in-list-prop-verifiedD watch-nat-lists-disjointD}$
 $\text{no-dup-filter-diff}[OF \ n\text{-d}] \text{ simp: } H$)[]
using single-nil **apply** simp
using single-other
apply ($\text{auto dest: filter-in-list-prop-verifiedD watch-nat-lists-disjointD}$
 $\text{no-dup-filter-diff}[OF \ n\text{-d}] \text{ simp: } H$)[]
using single-other **apply** ($\text{auto dest: filter-in-list-prop-verifiedD watch-nat-lists-disjointD}$
 $\text{no-dup-filter-diff}[OF \ n\text{-d}] \text{ simp: } H$)[]
using other $xs\text{-def}$ $ys\text{-def}$ **by** ($\text{metis } H$) +
qed

lemma $\text{watch-nat-lists-set-union}$:

fixes $C :: 'v::\text{linorder literal multiset}$ **and** $S :: ('v, 'a, 'b) \text{ twl-state}$
defines
 $xs \equiv [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \text{ . } - L \notin \text{lits-of } (\text{trail } S)]$ **and**
 $ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) \text{ . } L \in \# C]$
assumes $n\text{-d}$: $\text{no-dup } (\text{trail } S)$
shows $\text{set-mset } C = \text{set } xs \cup \text{set } ys$
using $n\text{-d}$ **unfolding** $xs\text{-def}$ $ys\text{-def}$ **by** ($\text{auto simp: lits-of-def uminus-lit-swap}$)

definition

$\text{rewatch-nat} ::$
 $(\text{nat}, \text{nat}, \text{nat literal multiset}) \text{ marked-lit} \Rightarrow (\text{nat}, \text{nat}, \text{nat clause}) \text{ twl-state} \Rightarrow$
 $\text{nat twl-clause} \Rightarrow \text{nat twl-clause}$

where

$\text{rewatch-nat } L \ S \ C =$
 $(\text{if } - \text{lit-of } L \in \# \text{ watched } C \text{ then}$
 $\text{case filter } (\lambda L'. L' \notin \# \text{ watched } C \wedge - L' \notin \text{lits-of } (L \# \text{trail } S))$

```

      (sorted-list-of-multiset (unwatched C)) of
    [] ⇒ C
  | L' # - ⇒
    TWL-Clause (watched C - {#- lit-of L#} + {#L'#}) (unwatched C - {#L'#} + {#- lit-of
L#})
    else
      C)

```

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*:
assumes no-dup (trail S)
shows raw-clause (watch-nat S C) = C
using assms
apply (cases rule: watch-nat-list-cases[OF assms(1), of C])
by (auto dest: filter-in-list-prop-verifiedD simp: watch-nat-def Let-def
mset-intersection-inclusion subseteq-mset-def)

lemma *distinct-pull[simp]*: $\text{distinct } (\text{pull } p \text{ } xs) = \text{distinct } xs$
unfolding pull-def **by** (induct xs) auto

lemma *falsified-watched-imp-unwatched-falsified*:
assumes
 watched: $L \in \text{set } (\text{take } n \text{ } (\text{pull } (\text{Not } \circ \text{fls}) \text{ } (\text{sorted-list-of-set } (\text{set-mset } C))))$ **and**
 falsified: $\text{fls } L$ **and**
 not-watched: $L' \notin \text{set } (\text{take } n \text{ } (\text{pull } (\text{Not } \circ \text{fls}) \text{ } (\text{sorted-list-of-set } (\text{set-mset } C))))$ **and**
 unwatched: $L' \in\# C - \text{mset } (\text{take } n \text{ } (\text{pull } (\text{Not } \circ \text{fls}) \text{ } (\text{sorted-list-of-set } (\text{set-mset } C))))$
shows $\text{fls } L'$

proof –
let ?Ls = sorted-list-of-set (set-mset C)
let ?W = take n (pull (Not ◦ fls) ?Ls)

have $n > \text{length } (\text{filter } (\text{Not } \circ \text{fls}) \text{ } ?Ls)$
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 *set-mset-is-single-in-mset-is-single*:
 $\text{set-mset } C = \{a\} \implies x \in\# C \implies x = a$
by fastforce

lemma *index-uminus-index-map-uminus*:
 $-a \in \text{set } L \implies \text{index } L \text{ } (-a) = \text{index } (\text{map } \text{uminus } L) \text{ } (a::'a \text{ literal})$
by (induction L) auto


```

lemma index-filter:
   $a \in \text{set } L \implies b \in \text{set } L \implies P a \implies P b \implies$ 
   $\text{index } L a \leq \text{index } L b \iff \text{index } (\text{filter } P L) a \leq \text{index } (\text{filter } P L) b$ 
  by (induction L) auto

lemma wf-watch-nat:  $\text{no-dup } (\text{trail } S) \implies \text{wf-twl-cls } (\text{trail } S) (\text{watch-nat } S C)$ 
  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)
proof goal-cases
  case 1
  then show ?case
    by (cases rule: watch-nat-list-cases[of S C]) (auto dest: filter-in-list-prop-verifiedD
      simp: distinct-mset-add-single)
next
  case 2
  then show ?case by simp
next
  case 3
  then show ?case
    proof (cases rule: watch-nat-list-cases[of S C])
      case nil-nil
      then have set-mset C = set [] ∪ set []
        using 3 by (metis watch-nat-lists-set-union)
      then show ?thesis
        by simp
    next
      case nil-single
      then show ?thesis
        using watch-nat-lists-set-union[of S C] 3 by(auto dest!: arg-cong[of - [] set])
    next
      case nil-other
      then show ?thesis
        using 3 by (auto dest!: arg-cong[of - [] set])
    next
      case single-nil
      show ?thesis
        using watch-nat-lists-set-union[of S C] 3 mset-leD unfolding single-nil by auto
    next
      case single-other
      then show ?thesis
        using 3 by (auto dest!: arg-cong[of - [] set])
    next
      case other
      then show ?thesis
        using 3 by (auto dest!: arg-cong[of - [] set])[]
    qed
  next
  case 4 note -[simp] = this
  {
    fix a :: nat literal and ys' :: nat literal list and L :: nat literal and
      L' :: nat literal
    assume a1: [L ← remdups (insort L (sorted-list-of-set (insert a (set ys') - {L})))]
      - L ∉ lits-of (trail S)] = [a]
  }

```

```

assume a2: set-mset C = insert L (insert a (set ys'))
assume a3:  $L' \in \# C$ 
assume a4:  $a \neq L'$ 
have set (L # a # ys') = set-mset C
  using a2 by auto
then have  $L' \notin \text{set } [l \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) . - l \notin \text{lits-of } (\text{trail } S)]$ 
  using a4 a1 by (metis List.finite-set list.set(1) list.set(2) singleton-iff
    sorted-list-of-set.insert-remove)
then have  $- L' \in \text{lits-of } (\text{trail } S)$ 
  using a3 by simp
} note H = this
show ?case using 4
  apply (cases rule: watch-nat-list-cases[of S C])
  apply (auto dest: filter-in-list-prop-verifiedD H simp: filter-empty-conv)[3]
  using watch-nat-lists-set-union[of S C] by (auto dest: filter-in-list-prop-verifiedD H)
next
case 5
then show ?case
  proof (cases rule: watch-nat-list-cases[of S C])
    case nil-nil
      then show ?thesis by auto
  next
    case nil-single
      then show ?thesis
        using watch-nat-lists-set-union[of S C] 5 by auto
  next
    case nil-other
      then show ?thesis
        unfolding watched-decided-most-recently.simps Ball-mset-def
        apply (intro allI impI)
        apply (subst index-uminus-index-map-uminus,
          simp add: index-uminus-index-map-uminus lits-of-def o-def)
        apply (subst index-uminus-index-map-uminus,
          simp add: index-uminus-index-map-uminus lits-of-def o-def)

        apply (subst index-filter[of - -  $\lambda L. L \in \# C$ ])
        by (auto dest: filter-in-list-prop-verifiedD
          simp: uminus-lit-swap lits-of-def o-def)
  next
    case single-nil
      then show ?thesis
        using watch-nat-lists-set-union[of S C] 5 by auto
  next
    case single-other
      then show ?thesis
        unfolding watched-decided-most-recently.simps Ball-mset-def
        apply (clarify)
        apply (subst index-uminus-index-map-uminus,
          simp add: index-uminus-index-map-uminus lits-of-def o-def)
        apply (subst index-uminus-index-map-uminus,
          simp add: index-uminus-index-map-uminus lits-of-def o-def)

        apply (subst index-filter[of - -  $\lambda L. L \in \# C$ ])
        by (auto dest: filter-in-list-prop-verifiedD simp: uminus-lit-swap lits-of-def o-def)
  next

```

```

case other
then show ?thesis
  apply clarsimp
  apply (elim disjE)
  prefer 2 apply (auto dest: filter-in-list-prop-verifiedD)[]
  apply (subst index-uminus-index-map-uminus,
    simp add: index-uminus-index-map-uminus lits-of-def o-def)[1]
  apply (subst index-uminus-index-map-uminus,
    simp add: index-uminus-index-map-uminus lits-of-def o-def)[1]

  apply (subst index-filter[of - - λL. L ∈# C])
  by (auto dest: filter-in-list-prop-verifiedD
    simp: index-uminus-index-map-uminus lits-of-def o-def uminus-lit-swap)
qed
qed

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: raw-clause (rewatch-nat L S C) = 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 size-mset-le-2-cases:
  assumes  $\text{size } W \leq 2$ 
  shows  $W = \{\#\} \vee (\exists a. W = \{\#a\}) \vee (\exists a\ b. W = \{\#a, b\})$ 
  by (metis One-nat-def Suc-1 Suc-eq-plus1-left assms linorder-not-less nat-less-le)

```

*not-less-eq-eq ordered-cancel-comm-monoid-diff-class.le-iff-add size-1-singleton-mset
size-eq-0-iff-empty size-mset-2)*

lemma *wf-rewatch-nat'*:

assumes

wf: *wf-twl-cls* (*trail S*) *C* **and**

n-d: *no-dup* (*trail S*) **and**

undef: *undefined-lit* (*trail S*) (*lit-of L*)

shows *wf-twl-cls* (*L # trail S*) (*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-nonfalsified* =

[L' ← sorted-list-of-multiset (unwatched C) . L' ∉ # watched C ∧ – L' ∉ lits-of (L # trail S)]

obtain *W UW* **where** *C*: *C = TWL-Clause W UW*

by (*cases C*)

show *?thesis*

proof (*cases ?unwatched-nonfalsified*)

case Nil

show *?thesis*

unfolding *rewatch-nat-def*

using *falsified Nil*

apply (*simp only: wf-twl-cls.simps if-True list.cases C*)

apply (*intro conjI*)

proof *goal-cases*

case 1

then show *?case* **using** *wf C* **by** *simp*

next

case 2

then show *?case* **using** *wf C* **by** *simp*

next

case 3

then show *?case* **using** *wf C* **by** *simp*

next

case 4

then show *?case* **using** *wf C* **by** *auto*

next

case 5

then show *?case*

using *C* **apply** *simp*

using *wf* **by** (*smt ball-msetI bspec-mset not-gr0 uminus-of-uminus-id
watched-decided-most-recently.simps wf-twl-cls.simps*)

qed

next

case (*Cons L' Ls*)

show *?thesis*

unfolding *rewatch-nat-def C*

using *falsified Cons*

apply (*simp only: wf-twl-cls.simps if-True list.cases C*)

apply (*intro conjI*)

proof *goal-cases*

case 1

then show *?case* **using** *wf C n-d*

```

    by (smt Multiset.diff-le-self distinct-mset-add-single distinct-mset-single-add
        filter-sorted-list-of-multiset-ConsD insert-DiffM mset-leD twl-clause.sel(1)
        wf-twlccls.simps)
next
case 2
then show ?case using wf C by (metis insert-DiffM2 size-single size-union twl-clause.sel(1)
    wf-twlccls.simps)
next
case 3
then show ?case
    using wf C by (force simp: mset-minus-single-eq-mempty dest: subset-singletonD)
next
case 4
have H:  $\forall L \in \# W. \neg L \in \text{ lits-of } (\text{trail } S) \longrightarrow$ 
    ( $\forall L' \in \# UW. \text{ count } W L' = 0 \longrightarrow \neg L' \in \text{ lits-of } (\text{trail } S)$ )
    using wf by (auto simp: C)
have W:  $\text{ size } W \leq 2$  and W-UW:  $\text{ size } W < 2 \longrightarrow \text{ set-mset } UW \subseteq \text{ set-mset } W$ 
    using wf by (auto simp: C)

have distinct: distinct-mset W
    using wf by (auto simp: C)
show ?case
    using 4
    unfolding C watched-decided-most-recently.simps Ball-mset-def twl-clause.sel
    apply (intro allI impI)
    apply (rename-tac xW xUW)
    apply (case-tac  $\neg \text{ lit-of } L = xW$ ; case-tac  $xW = xUW$ ; case-tac  $L' = xW$ )
        apply (auto simp: uminus-lit-swap)[2]
        using filter-sorted-list-of-multiset-ConsD apply blast
        using H size-mset-le-2-cases[OF W]
        using distinct apply (fastforce split: split-if-asm simp: distinct-mset-size-2)
        using distinct apply (fastforce split: split-if-asm simp: distinct-mset-size-2)
        using distinct apply (fastforce split: split-if-asm simp: distinct-mset-size-2)
        using filter-sorted-list-of-multiset-ConsD apply blast
    using size-mset-le-2-cases[OF W] H by (fastforce simp: uminus-lit-swap
        dest: filter-sorted-list-of-multiset-ConsD filter-sorted-list-of-multiset-eqD)

next
case 5
have H:  $\forall x. x \in \# W \longrightarrow \neg x \in \text{ lits-of } (\text{trail } S) \longrightarrow (\forall x. x \in \# UW \longrightarrow \text{ count } W x = 0$ 
     $\longrightarrow \neg x \in \text{ lits-of } (\text{trail } S))$ 
    using wf by (auto simp: C)

show ?case
    using 5 unfolding C watched-decided-most-recently.simps Ball-mset-def
    apply (intro allI impI conjI)
    apply (rename-tac xW x)
    apply (case-tac  $\neg \text{ lit-of } L = xW$ ; case-tac  $xW = x$ )
        apply (auto simp: uminus-lit-swap)[3]
    apply (case-tac  $\neg \text{ lit-of } L = x$ )
    apply (clarsimp)
    using H apply (blast dest: filter-sorted-list-of-multiset-ConsD
        filter-sorted-list-of-multiset-eqD)
    apply (clarsimp)
    using H apply (blast dest: filter-sorted-list-of-multiset-ConsD

```

```

      filter-sorted-list-of-multiset-eqD)
    done
  qed
qed
next
case False
then have wf-twl-cls (L # trail S) C
  apply (cases C)
  using wf n-d undef apply (clarify)
  unfolding wf-twl-cls.simps
  apply (intro conjI)
    apply blast
    apply blast
    apply blast
  apply (smt ball-mset-cong bspec-mset insert-iff lits-of-cons nat-neq-iff twl-clause.sel(1)
    uminus-of-uminus-id)
  apply (auto simp: Marked-Propagated-in-iff-in-lits-of)
done
then show ?thesis
  unfolding rewatch-nat-def using False by simp
qed

```

interpretation *twl*: *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)
done

```

Lifting to the abstract state.

```

context abstract-twl
begin

```

interpretation *state_W trail raw-init-clss raw-learned-clss backtrack-lvl conflicting*
*cons-trail tl-trail add-init-cl*s *add-learned-cl*s *remove-cl*s *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

```

interpretation *cdcl_W-ops trail raw-init-clss raw-learned-clss backtrack-lvl conflicting*
*cons-trail tl-trail add-init-cl*s *add-learned-cl*s *remove-cl*s *update-backtrack-lvl*
update-conflicting init-state restart'
 by *unfold-locales*

interpretation *cdcl_{NOT}*: *cdcl_{NOT}-merge-bj-learn-ops*
 $\lambda S.$ *convert-trail-from-W* (*trail S*)
clauses

$\lambda L S. \text{cons-trail } (\text{convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S$
 $\lambda L S. \text{lit-of } L \in \text{fst 'candidates-propagate } S$
 $\lambda S. \text{conflicting } S = C\text{-True}$
 $\lambda C C' L' S. C \in \text{candidates-conflict } S \wedge \text{distinct-mset } (C' + \{\#L'\# \}) \wedge \neg \text{tautology } (C' + \{\#L'\# \})$
by *unfold-locales*

interpretation *cdcl_{NOT}: cdcl_{NOT}-merge-bj-learn-proxy*

$\lambda S. \text{convert-trail-from-} W (\text{trail } S)$
clauses
 $\lambda L S. \text{cons-trail } (\text{convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S$
 $\lambda L S. \text{lit-of } L \in \text{fst 'candidates-propagate } S$
 $\lambda S. \text{conflicting } S = C\text{-True}$
 $\lambda C C' L' S. C \in \text{candidates-conflict } S$
apply *unfold-locales*
oops

declare *state-simp[simp del]*

abbreviation *cons-trail-twl where*

cons-trail-twl $L S \equiv \text{twl-of-rough-state } (\text{cons-trail } L (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-cons-trail:*

$\text{undefined-lit } (\text{trail } S) (\text{lit-of } L) \implies \text{wf-twl-state } S \implies \text{wf-twl-state } (\text{cons-trail } L S)$
unfolding *wf-twl-state-def* **by** (*auto simp: cons-trail-def wf-rewatch defined-lit-map*)

lemma *rough-state-of-twl-cons-trail:*

$\text{undefined-lit } (\text{trail-twl } S) (\text{lit-of } L) \implies$
 $\text{rough-state-of-twl } (\text{cons-trail-twl } L S) = \text{cons-trail } L (\text{rough-state-of-twl } S)$
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 $C S \equiv \text{twl-of-rough-state } (\text{add-init-cls } C (\text{rough-state-of-twl } S))$

lemma *wf-twl-add-init-cls: wf-twl-state* $S \implies \text{wf-twl-state } (\text{add-init-cls } L S)$

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:*

$\text{rough-state-of-twl } (\text{add-init-cls-twl } L S) = \text{add-init-cls } L (\text{rough-state-of-twl } S)$
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 $C S \equiv \text{twl-of-rough-state } (\text{add-learned-cls } C (\text{rough-state-of-twl } S))$

lemma *wf-twl-add-learned-cls: wf-twl-state* $S \implies \text{wf-twl-state } (\text{add-learned-cls } L S)$

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:*

$\text{rough-state-of-twl } (\text{add-learned-cls-twl } L S) = \text{add-learned-cls } L (\text{rough-state-of-twl } S)$
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 $C\ S \equiv \text{twl-of-rough-state } (\text{remove-cls } C\ (\text{rough-state-of-twl } S))$

lemma *wf-twl-remove-cls*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{remove-cls } L\ S)$

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 $N \equiv \text{twl-of-rough-state } (\text{init-state } N)$

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* $\square \ \{\#\} \ \{\#\} \ 0\ C\ \text{True}$)

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 $S \equiv \text{twl-of-rough-state } (\text{tl-trail } (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-tl-trail*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{tl-trail } S)$

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 $k\ S \equiv \text{twl-of-rough-state } (\text{update-backtrack-lvl } k\ (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-update-backtrack-lvl*:

$\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{update-backtrack-lvl } k\ S)$

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_W-twl-NOT*: *dpll-state*

AS. convert-trail-from-W (*trail-twl* S)

raw-clauses-twl

λL S. cons-trail-twl (*convert-marked-lit-from-NOT* L) S

λS. tl-trail-twl S

λC S. add-learned-cl C S

λC S. remove-cl C S

apply *unfold-locales*

apply (*simp add: rough-state-of-twl-cons-trail*)

apply (*metis rough-state-of-twl-tl-trail tl-trail*)

apply (*metis rough-state-of-twl-add-learned-cl* *trail-add-cl*_{NOT})

apply (*metis 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*_{NOT} *rough-state-of-twl-remove-cl* **by** *presburger*

interpretation *cdcl_W-twl*: *state_W*

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_W-twl: cdcl_W-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 $S S' \equiv \text{state-eq } (\text{rough-state-of-twl } S) (\text{rough-state-of-twl } S')$

notation *cdcl_W-twl.state-eq* (**infix** \sim 51)

declare *cdcl_W-twl.state-simp*[*simp del*]

cdcl_W-twl.state-simp_{NOT}[*simp del*]

cdcl_W-twl-NOT.state-simp_{NOT}[*simp del*]

To avoid ambiguities:

no-notation *CDCL-Two-Watched-Literals.twl.state-eq-twl* (**infix** $\sim TWL$ 51)

definition *propagate-twl* **where**

propagate-twl $S S' \longleftrightarrow$

$(\exists L C. (L, C) \in \text{candidates-propagate-twl } S$
 $\wedge S' \sim TWL \text{ cons-trail-twl } (\text{Propagated } L C) S$
 $\wedge \text{conflicting-twl } S = C\text{-True})$

lemma *propagate-twl-iff-propagate:*

assumes *inv: cdcl_W-twl.cdcl_W-all-struct-inv* S

shows *cdcl_W-twl.propagate* $S T \longleftrightarrow \text{propagate-twl } S T$ (**is** $?P \longleftrightarrow ?T$)

proof

assume $?P$

then obtain $C L$ **where**

conflicting $(\text{rough-state-of-twl } S) = C\text{-True}$ **and**

CL-Clauses: $C + \{\#L\# \} \in \# \text{ cdcl}_W\text{-twl.clauses } S$ **and**

```

S \models_{as} CNot\ C and
undef-lot: undefined-lit (trail-twl  $S$ )  $L$  and
 $T \sim cons\text{-}trail\text{-}twl\ (Propagated\ L\ (C + \{\#L\#\}))\ S$ 
unfolding  $cdcl_W\text{-}twl.propagate.simps$  by blast
have  $distinct\text{-}mset\ (C + \{\#L\#\})$ 
using inv  $CL\text{-}Clauses$  unfolding  $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ 
 $cdcl_W\text{-}twl.distinct\text{-}cdcl_W\text{-}state\text{-}def$   $cdcl_W\text{-}twl.clauses\text{-}def$   $distinct\text{-}mset\text{-}set\text{-}def$ 
by (metis (no-types, lifting) add-gr-0 mem-set-mset-iff plus-multiset.rep-eq)
then have  $C\text{-}L\text{-}L$ :  $mset\text{-}set\ (set\text{-}mset\ (C + \{\#L\#\}) - \{L\}) = C$ 
by (metis Un-insert-right add-diff-cancel-left' add-diff-cancel-right'
 $distinct\text{-}mset\text{-}set\text{-}mset\text{-}ident$   $finite\text{-}set\text{-}mset$   $insert\text{-}absorb2$   $mset\text{-}set.insert\text{-}remove$ 
 $set\text{-}mset\text{-}single$   $set\text{-}mset\text{-}union$ )
have  $(L, C + \{\#L\#\}) \in candidates\text{-}propagate\text{-}twl\ S$ 
apply (rule wf-candidates-propagate-complete)
using rough-state-of-twl apply auto[]
using  $CL\text{-}Clauses$  unfolding  $cdcl_W\text{-}twl.clauses\text{-}def$  apply auto[]
apply simp
using  $C\text{-}L\text{-}L$  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:  $\langle (L, C + \{\#L\#\}) \in candidates\text{-}propagate\text{-}twl\ S \rangle$ 
 $\langle conflicting\ (rough\text{-}state\text{-}of\text{-}twl\ S) = C\text{-}True \rangle$ )
using  $\langle T \sim cons\text{-}trail\text{-}twl\ (Propagated\ L\ (C + \{\#L\#\}))\ S \rangle$   $cdcl_W\text{-}twl.state\text{-}eq\text{-}backtrack\text{-}lvl$ 
 $cdcl_W\text{-}twl.state\text{-}eq\text{-}conflicting$   $cdcl_W\text{-}twl.state\text{-}eq\text{-}init\text{-}clss$ 
 $cdcl_W\text{-}twl.state\text{-}eq\text{-}learned\text{-}clss$   $cdcl_W\text{-}twl.state\text{-}eq\text{-}trail$  state-eq-def by blast
next
assume  $?T$ 
then obtain  $L\ C$  where
 $LC$ :  $(L, C) \in candidates\text{-}propagate\text{-}twl\ S$  and
 $T$ :  $T \sim TWL\ cons\text{-}trail\text{-}twl\ (Propagated\ L\ C)\ S$  and
 $confl$ :  $conflicting\ (rough\text{-}state\text{-}of\text{-}twl\ S) = C\text{-}True$ 
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
 $multi\text{-}member\text{-}last$  not-gr0 zero-diff)
have  $C \in \# raw\text{-}clauses\text{-}twl\ S$ 
using  $LC$  unfolding candidates-propagate-def clauses-def by auto
then have  $distinct\text{-}mset\ C$ 
using inv unfolding  $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$   $cdcl_W\text{-}twl.distinct\text{-}cdcl_W\text{-}state\text{-}def$ 
 $cdcl_W\text{-}twl.clauses\text{-}def$   $distinct\text{-}mset\text{-}set\text{-}def$  clauses-def by auto
then have  $C\text{-}L\text{-}L$ :  $mset\text{-}set\ (set\text{-}mset\ C - \{L\}) = C - \{\#L\#\}$ 
by (metis  $\langle C - \{\#L\#\} + \{\#L\#\} = C \rangle$  add-left-imp-eq diff-single-trivial
 $distinct\text{-}mset\text{-}set\text{-}mset\text{-}ident$   $finite\text{-}set\text{-}mset$   $mem\text{-}set\text{-}mset\text{-}iff$   $mset\text{-}set.remove$ 
 $multi\text{-}self\text{-}add\text{-}other\text{-}not\text{-}self$   $union\text{-}commute$ )

show  $?P$ 
apply (rule  $cdcl_W\text{-}twl.propagate.intros$ [of - trail-twl  $S$  init-clss-twl  $S$ 
 $learned\text{-}clss\text{-}twl\ S$  backtrack-lvl-twl  $S$   $C - \{\#L\#\}$   $L$ ])
using confl apply auto[]
using  $LC$  unfolding candidates-propagate-def apply (auto simp:  $cdcl_W\text{-}twl.clauses\text{-}def$ )[]
using wf-candidates-propagate-sound[OF -  $LC$ ] rough-state-of-twl apply (simp add:  $C\text{-}L\text{-}L$ )
using wf-candidates-propagate-sound[OF -  $LC$ ] rough-state-of-twl apply simp

```

using T unfolding $cdcl_W\text{-}twl.\text{state-eq-def}$ state-eq-def by auto
qed

term $local.\text{state-eq-twl}$

term $CDCL\text{-}Two\text{-}Watched\text{-}Literals.\text{twl}.\text{state-eq-twl}$

definition conflict-twl where

$\text{conflict-twl } S \ S' \longleftrightarrow$

$(\exists C. C \in \text{candidates-conflict-twl } S$
 $\wedge S' \sim \text{TWL update-conflicting-twl } (C\text{-Clause } C) \ S$
 $\wedge \text{conflicting-twl } S = C\text{-True})$

lemma $\text{conflict-twl-iff-conflict}$:

shows $cdcl_W\text{-}twl.\text{conflict } S \ T \longleftrightarrow \text{conflict-twl } S \ T$ (is $?C \longleftrightarrow ?T$)

proof

assume $?C$

then obtain $M \ N \ U \ k \ C$ where

S : $\text{state (rough-state-of-twl } S) = (M, N, U, k, C\text{-True})$ and

C : $C \in \# \text{ cdcl}_W\text{-}twl.\text{clauses } S$ and

$M\text{-}C$: $M \models_{as} C\text{Not } C$ and

T : $T \sim \text{update-conflicting-twl } (C\text{-Clause } C) \ S$

by auto

have $C \in \text{candidates-conflict-twl } S$

apply (rule $\text{wf-candidates-conflict-complete}$)

apply simp

using C apply (auto simp: $cdcl_W\text{-}twl.\text{clauses-def}$)[]

using $M\text{-}C \ S$ by auto

moreover have $T \sim \text{TWL twl-of-rough-state (update-conflicting } (C\text{-Clause } C) \ (\text{rough-state-of-twl } S))$

using T unfolding state-eq-def $cdcl_W\text{-}twl.\text{state-eq-def}$ by auto

ultimately show $?T$

using S unfolding conflict-twl-def by auto

next

assume $?T$

then obtain C where

C : $C \in \text{candidates-conflict-twl } S$ and

T : $T \sim \text{TWL update-conflicting-twl } (C\text{-Clause } C) \ S$ and

confl : $\text{conflicting-twl } S = C\text{-True}$

unfolding conflict-twl-def by auto

have $C \in \# \text{ cdcl}_W\text{-}twl.\text{clauses } S$

using C unfolding $\text{candidates-conflict-def}$ $cdcl_W\text{-}twl.\text{clauses-def}$ by auto

moreover have $\text{trail-twl } S \models_{as} C\text{Not } C$

using $\text{wf-candidates-conflict-sound}[OF - C]$ by auto

ultimately show $?C$ apply –

apply (rule $cdcl_W\text{-}twl.\text{conflict.conflict-rule}[of - - - - C]$)

using $\text{confl } T$ unfolding state-eq-def $cdcl_W\text{-}twl.\text{state-eq-def}$ by auto

qed

inductive $cdcl_W\text{-}twl :: 'v \text{ wf-twl} \Rightarrow 'v \text{ wf-twl} \Rightarrow \text{bool}$ for $S :: 'v \text{ wf-twl}$ where

propagate : $\text{propagate-twl } S \ S' \Longrightarrow cdcl_W\text{-}twl \ S \ S' \mid$

conflict : $\text{conflict-twl } S \ S' \Longrightarrow cdcl_W\text{-}twl \ S \ S' \mid$

other : $cdcl_W\text{-}twl.cdcl_W\text{-}o \ S \ S' \Longrightarrow cdcl_W\text{-}twl \ S \ S' \mid$

rf : $cdcl_W\text{-}twl.cdcl_W\text{-}rf \ S \ S' \Longrightarrow cdcl_W\text{-}twl \ S \ S'$

lemma $cdcl_W\text{-}twl\text{-}iff\text{-}cdcl_W$:

assumes $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv \ S$

shows $cdcl_W\text{-}twl \ S \ T \longleftrightarrow cdcl_W\text{-}twl.cdcl_W \ S \ T$

by (simp add: assms cdcl_W-twl.cdcl_W.simps cdcl_W-twl.simps conflict-tw-l-iff-conflict
propagate-tw-l-iff-propagate)

lemma rtrancpl-cdcl_W-twl-all-struct-inv-inv:
 assumes cdcl_W-twl** S T and cdcl_W-twl.cdcl_W-all-struct-inv S
 shows cdcl_W-twl.cdcl_W-all-struct-inv T
 using assms by (induction rule: rtrancpl-induct)
 (simp-all add: cdcl_W-twl-iff-cdcl_W cdcl_W-twl.cdcl_W-all-struct-inv-inv)

lemma rtrancpl-cdcl_W-twl-iff-rtrancpl-cdcl_W:
 assumes cdcl_W-twl.cdcl_W-all-struct-inv S
 shows cdcl_W-twl** S T \longleftrightarrow cdcl_W-twl.cdcl_W** S T (is ?T \longleftrightarrow ?W)

proof

assume ?W
 then show ?T
proof (induction rule: rtrancpl-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 cdcl_W-twl T U
 using assms st cdcl cdcl_W-twl.rtrancpl-cdcl_W-all-struct-inv-inv cdcl_W-twl-iff-cdcl_W
 by blast
 then show ?case using IH by auto
 qed
 next
 assume ?T
 then show ?W
proof (induction rule: rtrancpl-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 cdcl_W-twl.cdcl_W T U
 using assms st cdcl rtrancpl-cdcl_W-twl-all-struct-inv-inv cdcl_W-twl-iff-cdcl_W
 by blast
 then show ?case using IH by auto
 qed
 qed

interpretation cdcl_{NOT}-twl: backjumping-ops

λS. convert-trail-from-W (trail-tw-l S)
 abstract-tw-l.raw-clauses-tw-l
 λL (S:: 'v wf-tw-l).
 cons-trail-tw-l
 (convert-marked-lit-from-NOT L) (S:: 'v wf-tw-l)
 tl-trail-tw-l
 add-learned-cls-tw-l
 remove-cls-tw-l
 λC - - (S:: 'v wf-tw-l) -. C ∈ candidates-conflict-tw-l S
 by unfold-locales

lemma reduce-trail-to_{NOT}-skip-beginning-tw-l:
 assumes trail-tw-l S = convert-trail-from-NOT (F' @ F)
 shows trail-tw-l (cdcl_W-twl.reduce-trail-to_{NOT} F S) = convert-trail-from-NOT F

using *assms* **by** (*induction* F' *arbitrary*: S) *auto*

lemma *reduce-trail-to_{NOT}-trail-tl-trail-tw-l-decomp*[*simp*]:

$\text{trail-tw-l } S = \text{convert-trail-from-NOT } (F' @ \text{Marked } K () \# F) \implies$

$\text{trail-tw-l } (\text{cdcl}_W\text{-tw-l.reduce-trail-to}_{\text{NOT}} F (\text{tl-trail-tw-l } S)) = \text{convert-trail-from-NOT } F$

apply (*rule* *reduce-trail-to_{NOT}-skip-beginning-tw-l*[*of* - *tl* ($F' @ \text{Marked } K () \# []$)])

by (*cases* F') (*auto* *simp* *add:tl-append* *reduce-trail-to_{NOT}-skip-beginning*)

thm *tl-append* *reduce-trail-to_{NOT}-skip-beginning* *cdcl_W-tw-l.reduce-trail-to_{NOT}.sims*

lemma *trail-tw-l-reduce-trail-to_{NOT}-drop*:

$\text{trail-tw-l } (\text{cdcl}_W\text{-tw-l.reduce-trail-to}_{\text{NOT}} F S) =$

(*if* *length* (*trail-tw-l* S) \geq *length* F

then *drop* (*length* (*trail-tw-l* S) - *length* F) (*trail-tw-l* S)

else $[]$)

apply (*induction* F S *rule*: *cdcl_W-tw-l.reduce-trail-to_{NOT}.induct*)

apply (*rename-tac* F S)

apply (*case-tac* *trail-tw-l* S)

apply *auto* $[]$

apply (*rename-tac* *list*)

apply (*case-tac* *Suc* (*length* *list*) $>$ *length* F)

prefer 2 **apply** *simp*

apply (*subgoal-tac* *Suc* (*length* *list*) - *length* $F = \text{Suc } (\text{length } \text{list} - \text{length } F)$)

apply *simp*

apply *simp*

done

lemma *convert-trail-from-NOT-convert-trail-from-W*:

$(\forall l \in \text{set } F. \text{is-marked } l \wedge \text{level-of } l = 0 \vee \text{is-proped } l \wedge \text{mark-of } l = \{\#\}) \implies$

$\text{convert-trail-from-NOT } (\text{convert-trail-from-W } F) = F$

by (*induction* F *rule*: *marked-lit-list-induct*) *auto*

lemma *undefined-lit-convert-trail-from-NOT*[*simp*]:

$\text{undefined-lit } (\text{convert-trail-from-NOT } F) L \longleftrightarrow \text{undefined-lit } F L$

by (*induction* F *rule*: *marked-lit-list-induct*) (*auto* *simp*: *defined-lit-map*)

lemma *lits-of-convert-trail-from-NOT*:

$\text{lits-of } (\text{convert-trail-from-NOT } F) = \text{lits-of } F$

by (*induction* F *rule*: *marked-lit-list-induct*) *auto*

lemma *map-eq-cons-decomp*:

assumes SF : $\text{map } f l = xs @ ys$

shows $\exists xs' ys'. l = xs' @ ys' \wedge \text{map } f xs' = xs \wedge \text{map } f ys' = ys$

proof -

let $?F' = \text{take } (\text{length } xs) l$

let $?G = \text{drop } (\text{length } xs) l$

have $tr1$: $l = ?F' @ ?G$

by *simp*

moreover

have [*simp*]: $\text{length } l = \text{length } xs + \text{length } ys$

using *arg-cong*[*OF* SF , *of length*] **by** *auto*

have $\text{map } f ?F' = xs$ **and** $\text{map } f ?G = ys$

using *arg-cong*[*OF* SF , *of take* (*length* xs)] **apply** (*subst* (*asm*) $tr1$)

unfolding *map-append* **apply** *simp*

```

    using arg-cong[OF SF, of drop (length xs)] apply (subst (asm) tr1)
    unfolding map-append apply simp
  done
ultimately show ?thesis by blast
qed

```

interpretation *cdcl_{NOT}-twl: dpll-with-backjumping-ops*

```

λS. convert-trail-from-W (trail-tw1 S)
abstract-tw1.raw-clauses-tw1
λL (S:: 'v wf-tw1).
  cons-trail-tw1
  (convert-marked-lit-from-NOT L) (S:: 'v wf-tw1)
tl-trail-tw1
add-learned-clss-tw1
remove-clss-tw1
λL S. lit-of L ∈ fst ' candidates-propagate-tw1 S
λS. no-dup (trail-tw1 S)
  (* ∧ (∀ l ∈ set (trail-tw1 S). (is-marked l ∧ level-of l = 0) ∨ (is-proped l ∧ mark-of l = 0)) *)
λC - - (S:: 'v wf-tw1) -. C ∈ candidates-conflict-tw1 S

```

proof (*unfold-locales, goal-cases*)

```

case (1 C' S C F' K F L) note n-d = this(1) and n-d' = this(2) and undef = this(6)
let ?T' = (cons-trail (Propagated L {#}) (rough-state-of-tw1 (cdclW-tw1.reduce-trail-toNOT F S)))
let ?T = (cons-trail-tw1 (Propagated L {#}) (cdclW-tw1.reduce-trail-toNOT F S))
have tr-F-S: map lit-of (trail-tw1 (cdclW-tw1.reduce-trail-toNOT F S)) =
  map lit-of (convert-trail-from-NOT F)
  apply (subst trail-tw1.reduce-trail-toNOT-drop[of F S])
  using 1(1) arg-cong[OF 1(3), of length] arg-cong[OF 1(3), of map lit-of]
  by (auto simp: o-def drop-map[symmetric])

```

```

have no-dup (trail-tw1 S)
  using 1(1) by blast
have wf-tw1-state (rough-state-of-tw1 (cdclW-tw1.reduce-trail-toNOT F S))
  using wf-tw1-state-rough-state-of-tw1 by blast
moreover have undef': undefined-lit (trail-tw1 (cdclW-tw1.reduce-trail-toNOT F S)) L
  using undef arg-cong[OF tr-F-S, of map atm-of] unfolding defined-lit-map image-set
  by (simp add: o-def)
ultimately have wf-tw1-state ?T'
  by (simp-all add: wf-tw1-state-cons-trail)
then have init-clss-tw1 ?T = init-clss-tw1 (cdclW-tw1.reduce-trail-toNOT F S)
  using 1(6) by (simp add: undef')
then have [simp]: init-clss-tw1 ?T = init-clss-tw1 S
  by (simp add: cdclW-tw1.reduce-trail-toNOT-reduce-trail-convert)

```

```

have learned-clss-tw1 ?T = learned-clss-tw1 (cdclW-tw1.reduce-trail-toNOT F S)
  by (smt 1(3) 1(6) append-assoc cdclW-tw1.learned-clss-cons-trail
    cdclW-tw1-NOT.reduce-trail-toNOT-eq-length cdclW-tw1-NOT.reduce-trail-toNOT-nil
    cdclW-tw1-NOT.reduce-trail-toNOT-skip-beginning comp-apply defined-lit-convert-trail-from-W
    list.sel(3) marked-lit.sel(2) rev.simps(2) rev-append rev-eq-Cons-iff)
moreover have learned-clss-tw1 (cdclW-tw1.reduce-trail-toNOT F S)
  = learned-clss-tw1 S
  by (simp add: cdclW-tw1.reduce-trail-toNOT-reduce-trail-convert)
ultimately have [simp]: learned-clss-tw1 ?T = learned-clss-tw1 S
  by simp

```

```

have tr-L-F-S: map lit-of (trail-tw1 ?T) = map lit-of (Propagated L {#} # convert-trail-from-NOT

```

```

F)
  using undef' tr-F-S by (simp add: o-def)
  have C-conflict-cand:  $C \in \text{candidates-conflict-twl } S$ 
  apply (rule wf-candidates-twl-conflict-complete)
  using 1(1,4) apply (simp add: clauses-def)
  using 1(5) by (simp add: tr-L-F-S true-annots-true-cls lits-of-convert-trail-from-NOT)

  have cdclNOT-twl.backjump S
  (cons-trail-twl (convert-marked-lit-from-NOT (Propagated L ())))
  (cdclW-twl.reduce-trail-toNOT F S))
  apply (rule cdclNOT-twl.backjump.intros[of S F' K F - L C, OF 1(3) - 1(4-6) - 1(8-9)])
  unfolding cdclW-twl-NOT.state-eqNOT-def apply (metis convert-marked-lit-from-NOT.simps(1))
  using 1(7) 1(3) apply presburger
  using C-conflict-cand by simp
  then show ?case
  by blast
qed

interpretation cdclNOT-twl: dpll-with-backjumping
   $\lambda S.$  convert-trail-from-W (trail-twl S)
  abstract-twl.raw-clauses-twl
   $\lambda L (S:: 'v \text{ wf-twl}).$ 
  cons-trail-twl
  (convert-marked-lit-from-NOT L) (S:: 'v wf-twl)
  tl-trail-twl
  add-learned-cls-twl
  remove-cls-twl
   $\lambda L S.$  lit-of L  $\in$  fst 'candidates-propagate-twl S
   $\lambda S.$  no-dup (trail-twl S)
  ( $*$   $\wedge$  ( $\forall l \in \text{set (trail-twl S). (is-marked } l \wedge \text{level-of } l = 0) \vee (is-proped } l \wedge \text{mark-of } l = 0)$ ))  $*$ )
   $\lambda C - - (S:: 'v \text{ wf-twl}) - . C \in \text{candidates-conflict-twl } S$ 
  apply unfold-locales
  using cdclNOT-twl.dpll-bj-no-dup by (simp add: o-def)

end

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:
 $S \models_h C \longleftrightarrow \{\text{Pos } P | P. P \in S\} \cup \{\text{Neg } P | P. P \notin S\} \models C$ 
  unfolding Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def
  by auto

lemma herbrand-consistent-interp:
  consistent-interp ( $\{\text{Pos } P | P. P \in S\} \cup \{\text{Neg } P | P. P \notin S\}$ )
  unfolding consistent-interp-def by auto

```


lemma *herbrand-total-over-set*:

total-over-set ($\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$) *T*

unfolding *total-over-set-def* **by** *auto*

lemma *herbrand-total-over-m*:

total-over-m ($\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$) *T*

unfolding *total-over-m-def* **by** (*auto simp add: herbrand-total-over-set*)

lemma *herbrand-interp-iff-partial-interp-clss*:

$S \models_{hs} C \iff \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models_s C$

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: $\bigwedge C\ L. L \in \# S\ C \implies is_neg\ L$

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 δ , 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 Pos A = Max (set-mset C) \wedge count C (Pos A) \leq 1 \wedge \neg
interp C \models_h C \wedge S C = {#}

unfolding *production-unfold* **by** *auto*

lemma *produces* C A \implies 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 *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 *produces* C (*atm-of* (Max (set-mset C)))

unfolding *production-unfold* **by** (*auto* *simp* *del*: *atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom*: *productive* C \implies *produces* C (Max (*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 = *atm-of* (Max (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 = Max (*atms-of* C)

by (metis Max-singleton insert-not-empty productive-imp-produces-Max-atom)

lemma produces-imp-Pos-in-lits: produces C $A \implies Pos\ A \in\# C$
 by (auto intro: Max-in-lits dest!: producesD)

lemma productive-in-N: productive $C \implies C \in N$
 unfolding production-unfold by auto

lemma produces-imp-atms-leq: produces C $A \implies B \in atms-of\ C \implies B \leq A$
 by (metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom singleton-inject)

lemma produces-imp-neg-notin-lits: produces C $A \implies \neg Neg\ A \in\# C$
 by (auto intro!: pos-Max-imp-neg-notin dest: producesD simp del: not-gr0)

lemma less-eq-imp-interp-subseteq-interp: $C \# \subseteq\# D \implies interp\ C \subseteq interp\ D$
 unfolding interp-def by auto (metis multiset-order.order.strict-trans2)

lemma less-eq-imp-interp-subseteq-Interp: $C \# \subseteq\# D \implies interp\ C \subseteq Interp\ D$
 unfolding Interp-def using less-eq-imp-interp-subseteq-interp by blast

lemma less-imp-production-subseteq-interp: $C \# \subset\# D \implies production\ C \subseteq interp\ D$
 unfolding interp-def by fast

lemma less-eq-imp-production-subseteq-Interp: $C \# \subseteq\# D \implies production\ C \subseteq Interp\ D$
 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: $C \# \subset\# D \implies Interp\ C \subseteq interp\ D$
 unfolding Interp-def
 by (auto simp: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp)

lemma less-eq-imp-Interp-subseteq-Interp: $C \# \subseteq\# D \implies Interp\ C \subseteq Interp\ D$
 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: $A \notin Interp\ C \implies A \in interp\ D \implies C \# \subset\# D$
 using less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less by blast

lemma false-interp-to-true-interp-imp-less-multiset: $A \notin interp\ C \implies A \in interp\ D \implies C \# \subset\# D$
 using less-eq-imp-interp-subseteq-interp multiset-linorder.not-less by blast

lemma false-Interp-to-true-Interp-imp-less-multiset: $A \notin Interp\ C \implies A \in Interp\ D \implies C \# \subset\# D$
 using less-eq-imp-Interp-subseteq-Interp multiset-linorder.not-less by blast

lemma false-interp-to-true-Interp-imp-le-multiset: $A \notin interp\ C \implies A \in Interp\ D \implies C \# \subseteq\# D$
 using less-imp-Interp-subseteq-interp multiset-linorder.not-less by blast

lemma interp-subseteq-INTERP: $interp\ C \subseteq INTERP$
 unfolding interp-def INTERP-def by (auto simp: production-unfold)

lemma production-subseteq-INTERP: $production\ C \subseteq INTERP$
 unfolding INTERP-def using production-unfold by blast

lemma Interp-subseteq-INTERP: $Interp\ C \subseteq INTERP$

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** $Max (set-mset D) = Pos A$

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  $D: \{\#Pos\ P, Pos\ P\#\} \# \subset \# C$ 
shows  $\neg produces\ C\ P$ 
proof –
  let  $?D = \{\#Pos\ P, Pos\ P\#\}$ 
  note  $D' = D[unfolded\ less-multiset_{HO}]$ 
  consider
     $(P)\ count\ C\ (Pos\ P) \geq 2$ 
  |  $(Q)\ Q\ \textbf{where}\ Q > Pos\ P\ \textbf{and}\ Q \in \# C$ 
    using  $HOL.spec[OF\ HOL.conjunct2[OF\ D'],\ of\ Pos\ P]$  by auto
  thus ?thesis
  proof cases
    case  $Q$ 
    have  $Q \in set-mset\ C$ 
    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  $D: C + \{\#Neg\ P\#\} \# \subset \# D$ 
shows  $production\ D \neq \{P\}$ 
proof –
  note  $D' = D[unfolded\ less-multiset_{HO}]$ 
  consider
     $(P)\ Neg\ P \in \# D$ 
  |  $(Q)\ Q\ \textbf{where}\ Q > Neg\ P\ \textbf{and}\ count\ D\ Q > count\ (C + \{\#Neg\ P\#\})\ Q$ 
    using  $HOL.spec[OF\ HOL.conjunct2[OF\ D'],\ of\ Neg\ P]$  by fastforce
  thus ?thesis
  proof cases
    case  $Q$ 
    have  $Q \in set-mset\ D$ 
    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 $P \in \text{INTERP}$
shows $\exists D. D + \{\#Pos\ P\# \} \in N \wedge \text{produces } (D + \{\#Pos\ P\# \})\ P$
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: $A \in N \Longrightarrow B \in N \Longrightarrow \text{superposition-rules } A\ B\ C$
 $\Longrightarrow \text{superposition } N\ (N \cup \{C\})$

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 $IH: \bigwedge I. \text{total-over-set } I\ (\text{atms-of } C) \Longrightarrow \text{total-over-m } I\ B \Longrightarrow \text{consistent-interp } I$
 $\Longrightarrow I \models_s B \Longrightarrow I \models C$ **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
 $abstr: \text{abstract-red } C\ N$ **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 class-lt-def*
by (*metis (no-types, lifting) c-lt-d subset-mset.diff-add true-class-cls-mono-r'*
true-class-cls-subset)
qed

lemma *true-class-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 $?I = I \cup \{Pos\ P \mid P. P \in \text{atms-of } B \wedge P \notin \text{atms-of-s } I\}$
 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 $aa :: 'a \text{ set} \Rightarrow 'a \text{ literal set} \Rightarrow 'a$ **where**
 $f2: \forall x0\ x1. (\exists v2. v2 \in x0 \wedge Pos\ v2 \notin x1 \wedge Neg\ v2 \notin x1)$
 $\longleftrightarrow (aa\ x0\ x1 \in x0 \wedge Pos\ (aa\ x0\ x1) \notin x1 \wedge Neg\ (aa\ x0\ x1) \notin x1)$
by *moura*
have $\forall a. a \notin \text{atms-of-ms } A \vee Pos\ a \in I \vee Neg\ a \in I$
 using *tot by (simp add: total-over-m-def total-over-set-def)*
hence $aa\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})\ (I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})$
 $\notin \text{atms-of-ms } A \cup \text{atms-of-ms } \{B\} \vee Pos\ (aa\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\}))$
 $(I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})) \in I$
 $\cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}$
 $\vee Neg\ (aa\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\}))$
 $(I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})) \in I$
 $\cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}$
by *auto*
hence *total-over-set* $(I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})$
 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-class-cls-def* **by** *auto*
 thus *?thesis*

oops

lemma

assumes
 $CP: \neg \text{class-lt } N\ (\{\#C\# \} + \{\#E\# \}) \models_p \{\#C\# \} + \{\#Neg\ P\# \}$ **and**

$\text{clss-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#E\# \} + \{ \#Pos P\# \} \vee \text{clss-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#C\# \} + \{ \#Neg P\# \}$
shows $\text{clss-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#E\# \} + \{ \#Pos P\# \}$

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: $\text{redundant } A N \longleftrightarrow \text{abstract-red } A N$

begin

definition *saturated* :: 'a clauses \Rightarrow bool **where**

$\text{saturated } N \longleftrightarrow (\forall A B C. A \in N \longrightarrow B \in N \longrightarrow \neg \text{redundant } A N \longrightarrow \neg \text{redundant } B N \longrightarrow \text{superposition-rules } A B C \longrightarrow \text{redundant } C N \vee C \in N)$

lemma

assumes

saturated: $\text{saturated } N$ **and**

finite: $\text{finite } N$ **and**

empty: $\{ \# \} \notin N$

shows $\text{INTERP } N \models_{hs} N$

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 $D \equiv \text{Min } \{ E \in N. \neg ?N_{\mathcal{I}} \models_h E \}$

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** $LSD: L \in \# S D \vee (S D = \{ \# \} \wedge \text{Max } (\text{set-mset } D) = L)$

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 (D ∈ N) 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  $?N_{\mathcal{I}} \models_{hs} clss\text{-}lt$   $N$   $D$ 
    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  $L = Pos$   $P$  and  $S$   $D = \{\#\}$  and  $Max$  (set-mset  $D$ ) =  $Pos$   $P$ 
| ( $Lneg$ )  $P$  where  $L = Neg$   $P$ 
  using LSD  $S$ -selects-neg-lits[of  $D$   $L$ ] by (cases  $L$ ) auto
thus False
proof cases
  case  $L$  note  $P = this(1)$  and  $S = this(2)$  and  $max = this(3)$ 
  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  $C':D = C' + \{\#Pos$   $P\#\} + \{\#Pos$   $P\#\}$ 
  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  $C' + \{\#Pos$   $P\#\} \# \subset \# C' + \{\#Pos$   $P\#\} + \{\#Pos$   $P\#\}$ 
  by auto
moreover have  $\neg ?N_{\mathcal{I}} \models_h (D - \{\#L\#\})$ 
  using not-d-interp unfolding  $C'$   $L$  by auto
ultimately have  $C' + \{\#Pos$   $P\#\} \notin N$ 
  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 $c'-p-p$: $C' + \{\#Pos\ P\# \} + \{\#Pos\ P\# \} - \{\#Pos\ P\# \} = C' + \{\#Pos\ P\# \}$
by *auto*
have *redundant* $(C' + \{\#Pos\ P\# \})\ N$
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_{HO}* **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  $abs: clss\text{-}lt\ N\ (C + E) \models_p C + E$ 
  using redundant-iff-abstract red' unfolding abstract-red-def by auto
  have  $clss\text{-}lt\ N\ (C + E) \models_p E + \{\#Pos\ P\# \} \vee clss\text{-}lt\ N\ (C + E) \models_p C + \{\#Neg\ P\# \}$ 
  proof clarify
    assume  $CP: \neg clss\text{-}lt\ N\ (C + E) \models_p C + \{\#Neg\ P\# \}$ 
    { fix  $I$ 
      assume
        total-over-m  $I\ (clss\text{-}lt\ N\ (C + E) \cup \{E + \{\#Pos\ P\# \}\})$  and
        consistent-interp  $I$  and
         $I \models_s clss\text{-}lt\ N\ (C + E)$ 
      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  $clss\text{-}lt\ N\ (C + E) \models_p E + \{\#Pos\ P\# \}$ 
      unfolding true-clss-clss-def by auto
  qed
  moreover have  $clss\text{-}lt\ N\ (C + E) \subseteq clss\text{-}lt\ N\ (C + \{\#Neg\ P\# \})$ 
  using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
  ultimately have  $redundant\ (C + \{\#Neg\ P\# \})\ N \vee clss\text{-}lt\ N\ (C + E) \models_p E + \{\#Pos\ P\# \}$ 
  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  $A \in \text{class-lt } N \ B$  using  $AN \ AB$  unfolding class-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-class-cls-def Partial-Clausal-Logic.true-class-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  $A :: 'a :: \text{wellorder clause}$  and  $N :: 'a :: \text{wellorder clauses}$ 
  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 class-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  $S :: \text{nat clause} \Rightarrow \text{nat clause}$ 
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

```