

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

March 8, 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	11
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	62
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	66
9.2.1	Full CNF transformation	66
9.3	Disjunctive Normal Form	67
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	75
11.2.2	Atoms	76
11.2.3	Totality	78
11.2.4	Interpretations	80
11.2.5	Satisfiability	82
11.2.6	Entailment for Multisets of Clauses	83
11.2.7	Tautologies	85
11.2.8	Entailment for clauses and propositions	86
11.3	Subsumptions	91
11.4	Removing Duplicates	92
11.5	Set of all Simple Clauses	92
11.6	Experiment: Expressing the Entailments as Locales	95
11.7	Entailment to be extended	96
12	Resolution	97
12.1	Simplification Rules	97
12.2	Unconstrained Resolution	99
12.2.1	Subsumption	99
12.3	Inference Rule	100
12.4	Lemma about the simplified state	115
12.5	Resolution and Invariants	118
12.5.1	Invariants	118
12.5.2	well-foundedness if the relation	124
13	Partial Clausal Logic	138
13.1	Marked Literals	138
13.1.1	Definition	138
13.1.2	Entailment	140
13.1.3	Defined and undefined literals	141
13.2	Backtracking	143
13.3	Decomposition with respect to the marked literals	143

13.4	Negation of Clauses	150
13.5	Other	154
13.6	Abstract Clause Representation	155
14	Measure	157
15	NOT's CDCL	161
15.1	Auxiliary Lemmas and Measure	161
15.2	Initial definitions	161
15.2.1	The state	161
15.2.2	Definition of the operation	165
15.3	DPLL with backjumping	167
15.3.1	Definition	168
15.3.2	Basic properties	168
15.3.3	Termination	171
15.3.4	Normal Forms	176
15.4	CDCL	183
15.4.1	Learn and Forget	183
15.4.2	Definition of CDCL	185
15.5	CDCL with invariant	189
15.6	Termination	194
15.6.1	Restricting learn and forget	194
15.7	CDCL with restarts	206
15.7.1	Definition	206
15.7.2	Increasing restarts	207
15.8	Merging backjump and learning	215
15.8.1	Instantiations	228
16	DPLL as an instance of NOT	242
16.1	DPLL with simple backtrack	242
16.2	Adding restarts	248
17	DPLL	249
17.1	Rules	249
17.2	Invariants	249
17.3	Termination	257
17.4	Final States	260
17.5	Link with NOT's DPLL	261
17.5.1	Level of literals and clauses	262
17.5.2	Properties about the levels	266
18	Weidenbach's CDCL	269
18.1	The State	269
18.2	CDCL Rules	278
18.3	Invariants	285
18.3.1	Properties of the trail	285
18.3.2	Better-Suited Induction Principle	289
18.3.3	Compatibility with $op \sim$	293
18.3.4	Conservation of some Properties	298
18.3.5	Learned Clause	298

18.3.6	No alien atom in the state	300
18.3.7	No duplicates all around	303
18.3.8	Conflicts and co	304
18.3.9	Putting all the invariants together	313
18.3.10	No tautology is learned	316
18.4	CDCL Strong Completeness	316
18.5	Higher level strategy	318
18.5.1	Definition	318
18.5.2	Invariants	320
18.5.3	Literal of highest level in conflicting clauses	326
18.5.4	Literal of highest level in marked literals	330
18.5.5	Strong completeness	340
18.5.6	No conflict with only variables of level less than backtrack level	347
18.5.7	Final States are Conclusive	357
18.6	Termination	364
18.7	No Relearning of a clause	365
18.8	Decrease of a measure	381
19	Simple Implementation of the DPLL and CDCL	387
19.1	Common Rules	387
19.1.1	Propagation	387
19.1.2	Unit propagation for all clauses	388
19.1.3	Decide	390
19.2	Simple Implementation of DPLL	390
19.2.1	Combining the propagate and decide: a DPLL step	390
19.2.2	Adding invariants	393
19.2.3	Code export	400
19.3	CDCL Implementation	404
19.3.1	Definition of the rules	404
19.3.2	The Transitions	406
19.3.3	Code generation	418
20	Link between Weidenbach's and NOT's CDCL	433
20.1	Inclusion of the states	433
20.2	More lemmas conflict-propagate and backjumping	434
20.2.1	Termination	434
20.2.2	More backjumping	435
20.3	CDCL FW	437
20.4	FW with strategy	442
20.4.1	The intermediate step	442
20.5	Adding Restarts	477
21	Link between Weidenbach's and NOT's CDCL	488
21.1	Inclusion of the states	488
21.2	Additional Lemmas between NOT and W states	492
21.3	More lemmas conflict-propagate and backjumping	494
21.4	CDCL FW	494

22.1 We can now define the rules of the calculus 518
 theory *Wellfounded-More*
 imports *Main*

begin

1 Transitions

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

1.1 More theorems about Closures

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

lemma *trancpl-mono-explicit*:

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

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

lemma *trancpl-mono*:

assumes *mono*: $r \leq s$

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

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

lemma *trancpl-idemp-rel*:

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

apply (*rule iffI*)

prefer 2 **apply** *blast*

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

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

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

by *simp*

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

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

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

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

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

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

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

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

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

apply (*rule iffI*)

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

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

done

lemma *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 *assms(1) step.hyps(1) step.premis(2)* **apply** *auto[1]*
using $IH z$ *rtranclp.rtrancl-into-rtrancl* **by** *fastforce*
qed

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

1.2 Full Transitions

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

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

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

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

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

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

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

lemma *tranclp-full-full1I*:
 $R^{++} a b \Longrightarrow \text{full } R b c \Longrightarrow \text{full1 } R a c$
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]*: $\text{full1}\ R\ S\ T \implies \text{full}\ R\ S\ T$

by (*simp add: full-unfold*)

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

by (*meson full1-def rtranclp.rtrancl-refl*)

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

by (*meson full-full1 not-full1-rtranclp-relation rtranclp.rtrancl-refl*)

lemma *full1-tranclp-relation-full*:

$\text{full1}\ R^{++}\ a\ b \longleftrightarrow \text{full1}\ R\ a\ b$
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*:

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

lemma *rtranclp-full1-eq-or-full1*:

$(\text{full1}\ R)^{**}\ a\ b \longleftrightarrow (a = b \vee \text{full1}\ R\ a\ b)$

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 $aa :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **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 $\neg \text{full1}\ R\ a\ b \wedge a \neq b$

then have $a \neq b \wedge a \neq b \wedge \neg \text{full1}\ R\ (aa\ (\text{full1}\ R)\ a\ b)\ b \vee \neg (\text{full1}\ R)^{**}\ a\ b \wedge a \neq b$
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*:

$(\text{full1}\ R)^{++}\ a\ b \longleftrightarrow \text{full1}\ R\ a\ b$

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 $\exists b. R^{**}\ a\ b \wedge \text{no-step}\ R\ b$

proof (*rule ccontr*)

assume $\neg ?thesis$

```

then have  $H: \bigwedge b. \neg R^{**} a b \vee \neg \text{no-step } R b$ 
  by blast
def  $F \equiv \text{rec-nat } a (\lambda i b. \text{SOME } c. R b c)$ 
have [simp]:  $F 0 = a$ 
  unfolding  $F\text{-def}$  by auto
have [simp]:  $\bigwedge i. F (\text{Suc } i) = (\text{SOME } b. R (F i) b)$ 
  using  $F\text{-def}$  by simp
{ fix  $i$ 
  have  $\forall j < i. R (F j) (F (\text{Suc } j))$ 
    proof (induction  $i$ )
      case 0
      then show ?case by auto
    next
      case ( $\text{Suc } i$ )
      then have  $R^{**} a (F i)$ 
        by (induction  $i$ ) auto
      then have  $R (F i) (\text{SOME } b. R (F i) b)$ 
        using  $H$  by (simp add: someI-ex)
      then have  $\forall j < \text{Suc } i. R (F j) (F (\text{Suc } j))$ 
        using  $H \text{ Suc}$  by (simp add: less-Suc-eq)
      then show ?case by fast
    qed
  }
then have  $\forall j. R (F j) (F (\text{Suc } j))$  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  $\text{wf}:\text{wf } \{(x, y). R y x\}$ 
  shows  $\exists b. \text{full } R a b$ 
  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: $\text{wf } ?r = (\nexists f. \forall i. (f (\text{Suc } i), f i) \in ?r), \llbracket \text{wf } ?r; \bigwedge k. (?f (\text{Suc } k), ?f k) \notin ?r \implies ?thesis \rrbracket \implies ?thesis$

```

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

```

```

lemma wfP-if-measure: fixes  $f :: 'a \Rightarrow \text{nat}$ 
shows  $(\bigwedge x y. P x \implies g x y \implies f y < f x) \implies \text{wf } \{(y, x). P x \wedge g x y\}$ 
  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  $\text{wf } r$ 
  shows  $\text{wf } \{(b, a). (f b, f a) \in r\}$ 

```


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-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-subset* by *simp*

qed

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

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

apply(*insert wf-measure*[*of f*])

apply(*simp only: measure-def inv-image-def less-than-def less-eq*)

apply(*erule wf-subset*)

apply *auto*

done

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

assumes

P-finite: $\bigwedge U. P U \longrightarrow U \in A$ and

finite: *finite* *A* and

wf: *wf* *R* and

trans: *trans* *R*

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

proof (*rule wfP-if-measure2*)

fix *T S*

assume *P*: $P S \wedge P T$ and

s-le-t: $(T, S) \in \text{lexord } R$

let *?f* = $\lambda S. \{U. (U, S) \in \text{lexord } R \wedge P U \wedge P S\}$

have *?f T* \subseteq *?f S*

using *s-le-t P lexord-trans trans* by *auto*

moreover have $T \in ?f S$

using *s-le-t P* by *auto*

moreover have $T \notin ?f T$

using *s-le-t* by (*auto simp add: lexord-irreflexive local.wf*)

ultimately have $\{U. (U, T) \in \text{lexord } R \wedge P U \wedge P T\} \subset \{U. (U, S) \in \text{lexord } R \wedge P U \wedge P S\}$ by *auto*

moreover have *finite* $\{U. (U, S) \in \text{lexord } R \wedge P U \wedge P S\}$

using *finite* by (*metis (no-types, lifting) P-finite finite-subset mem-Collect-eq subsetI*)

ultimately show $\text{card } (?f T) < \text{card } (?f S)$ by (*simp add: psubset-card-mono*)

qed

lemma *wf-fst-wf-pair*:

```

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

lemma wf-snd-wf-pair:
  assumes wf  $\{(M', M). R M' M\}$ 
  shows wf  $\{((M', N'), (M, N)). R N' N\}$ 
  proof -
    have wf: wf  $\{((M', N'), (M, N)). R M' M\}$ 
      using assms wf-fst-wf-pair by auto
    then have wf:  $\bigwedge P. (\forall x. (\forall y. (y, x) \in \{((M', N'), M, N). R M' M\} \longrightarrow P y) \longrightarrow P x) \implies \text{All } P$ 
      unfolding wf-def by auto
    show ?thesis
      unfolding wf-def
    proof (intro allI impI)
      fix  $P :: 'c \times 'a \Rightarrow \text{bool}$  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  $\text{wf}[of\ P\ o\ (\lambda(a, b). (b, a))]$  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

Sledgehammer parameters

sledgehammer-params[*debug*]

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**

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

shows $P\ n$

apply (*induction rule: nat-less-induct*)

by (*rename-tac n, 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*]:

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

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 $f :: nat \Rightarrow nat$

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 $\bigwedge n. f\ n \leq Max\ (\{f\ n \mid n. n \leq n_0\} \cup \{m\})$

apply (*case-tac n ≤ n₀*)

apply (*metis (mono-tags, lifting) Max-ge Un-insert-right ⟨finite {f n | n. n ≤ n₀⟩*
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 | n. n ≤ n₀⟩*
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 $k :: nat$ **and** $f :: nat \Rightarrow nat$

assumes $k > 0$

```

shows bounded  $f \longleftrightarrow \text{bounded } (\lambda i. k * f i)$ 
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  $\bigwedge x. f x \leq \text{Max } \{f x | x. \text{True}\}$ 
    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 $\langle B = [Suc\ m + \text{length } A..<n] \rangle$ by *auto*
 qed

lemma *Max-n-upt*: $Max\ (insert\ 0\ \{Suc\ 0..<n\}) = n - Suc\ 0$
 proof (induct n)
 case 0
 then show $?case$ by *simp*
 next
 case (Suc n) note *IH* = *this*
 have i : $insert\ 0\ \{Suc\ 0..<Suc\ n\} = insert\ 0\ \{Suc\ 0..<n\} \cup \{n\}$ by *auto*
 show $?case$ using *IH* unfolding i by *auto*
 qed

lemma *upt-decomp-lt*:
 assumes H : $xs\ @\ i\ \# \ ys\ @\ j\ \# \ zs = [m\ ..<n]$
 shows $i < j$
 proof -
 have xs : $xs = [m\ ..<i]$ 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*:
 $[c, d] = ys\ @\ y\ \# \ ys' \longleftrightarrow (ys = [] \wedge y = c \wedge ys' = [d]) \vee (ys = [c] \wedge y = d \wedge ys' = [])$
 by (cases ys ; cases ys') *auto*

lemma *lexn2-conv*:
 $([a, b], [c, d]) \in \text{lexn}\ r\ 2 \longleftrightarrow (a, c) \in r \vee (a = c \wedge (b, d) \in r)$
 unfolding *lexn-conv* by (auto simp add: *list-length2-append-cons*)

end
 theory *Prop-Logic*

imports *Main*

begin

4 Logics

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

4.1 Definition and abstraction

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

datatype $'v\ \text{propo} =$
 $FT \mid FF \mid FVar\ 'v \mid FNot\ 'v\ \text{propo} \mid FAnd\ 'v\ \text{propo}\ 'v\ \text{propo} \mid FOr\ 'v\ \text{propo}\ 'v\ \text{propo}$

| *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*[*case-names nullary binary unary*]:

assumes *nullary*: $\bigwedge x. c = CT \vee c = CF \vee c = CVar\ x \implies P$
and *binary*: $c \in \text{binary-connectives} \implies P$
and *unary*: $c = CNot \implies P$
shows P
using *assms* **by** (*cases c*) (*auto simp: 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** (*cases 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]: ($c = CT \vee c = CF \vee c = CVar\ v$) \Longrightarrow *wf-conn* *c* [] |
wf-conn-unary[simp]: $c = CNot \Longrightarrow$ *wf-conn* *c* [ψ] |
wf-conn-binary[simp]: $c \in \text{binary-connectives} \Longrightarrow$ *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**
 ($\bigwedge v. c = CT \Longrightarrow P\ []$) **and**
 ($\bigwedge v. c = CF \Longrightarrow P\ []$) **and**
 ($\bigwedge v. c = CVar\ v \Longrightarrow P\ []$) **and**
 ($\bigwedge \psi. c = CNot \Longrightarrow P\ [\psi]$) **and**
 ($\bigwedge \psi\ \psi'. c = COr \Longrightarrow P\ [\psi, \psi']$) **and**
 ($\bigwedge \psi\ \psi'. c = CAnd \Longrightarrow P\ [\psi, \psi']$) **and**
 ($\bigwedge \psi\ \psi'. c = CImp \Longrightarrow P\ [\psi, \psi']$) **and**
 ($\bigwedge \psi\ \psi'. c = CEq \Longrightarrow P\ [\psi, \psi']$)
shows $P\ x$
using *assms* **by** *induction (auto simp: binary-connectives-def)*

4.2 properties of the abstraction

First we can define simplification rules.

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

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

lemma *wf-conn-list*:
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FT* $\longleftrightarrow (c = CT \wedge l = [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FF* $\longleftrightarrow (c = CF \wedge l = [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FVar* *x* $\longleftrightarrow (c = CVar\ x \wedge l = [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FAnd* *a* *b* $\longleftrightarrow (c = CAnd \wedge l = a \# b \# [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FOr* *a* *b* $\longleftrightarrow (c = COr \wedge l = a \# b \# [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FEq* *a* *b* $\longleftrightarrow (c = CEq \wedge l = a \# b \# [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FImp* *a* *b* $\longleftrightarrow (c = CImp \wedge l = a \# b \# [])$
wf-conn *c* *l* \Longrightarrow *conn* *c* *l* = *FNot* *a* $\longleftrightarrow (c = CNot \wedge l = a \# [])$
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 (rename-tac l, 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  $\in$  binary-connectives
  shows length l = 2  $\longleftrightarrow$  wf-conn c l
proof
  assume length l = 2
  then show wf-conn c l using wf-conn-binary list-length2-decomp using conn by metis
next
  assume wf-conn c l
  then show length l = 2 (is ?P l)
  proof (cases rule: wf-conn.induct)
    case wf-conn-nullary
    then show ?P [] using conn binary-connectives-def
      using connective.distinct(11) connective.distinct(13) connective.distinct(9) by blast
  next
    fix  $\psi$  :: 'v propo
    case wf-conn-unary
    then show ?P [ $\psi$ ] using conn binary-connectives-def
      using connective.distinct by blast
  next
    fix  $\psi$   $\psi'$  :: 'v propo
    show ?P [ $\psi$ ,  $\psi'$ ] by auto
  qed
qed

```

```

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

```

Decomposing the Not into an element is moreover very useful.

```

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

```

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

```

lemma wf-conn-no-arity-change:
  length l = length l'  $\implies$  wf-conn c l  $\longleftrightarrow$  wf-conn c l'
proof -
  {
    fix l l'
    have length l = length l'  $\implies$  wf-conn c l  $\implies$  wf-conn c l'
    apply (cases c l rule: wf-conn.induct, auto)
  }

```

```

    by (metis wf-conn-bin-list-length)
  }
  then show length l = length l'  $\implies$  wf-conn c l = wf-conn c l' by metis
qed

```

```

lemma wf-conn-no-arity-change-helper:
  length (ξ @ φ # ξ') = length (ξ @ φ' # ξ')
  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 ψ
  shows c = CNot and l = [ψ]
  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 ψs :: 'v propo list
  assumes corr: wf-conn ca l
  and corr': wf-conn c ψs
  and eq: conn ca l = conn c ψs
  shows ca = c ∧ ψs = l
  using corr
proof (cases ca l rule: wf-conn.cases)
  case (wf-conn-nullary v)
  then show ca = c ∧ ψs = l using assms
    by (metis conn.simps(1) conn.simps(2) conn.simps(3) wf-conn-list(1-3))
next
  case (wf-conn-unary ψ')
  then have *: FNot ψ' = conn c ψs using conn-inj-not eq assms by auto
  then have c = ca by (metis conn-inj-not(1) corr' wf-conn-unary(2))
  moreover have ψs = l using * conn-inj-not(2) corr' wf-conn-unary(1) by force
  ultimately show ca = c ∧ ψs = l by auto
next
  case (wf-conn-binary ψ' ψ'')
  then show ca = c ∧ ψs = l
    using eq corr' unfolding binary-connectives-def apply (cases 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]: φ  $\preceq$  φ |
  subformula-into-subformula: ψ ∈ set l  $\implies$  wf-conn c l  $\implies$  φ  $\preceq$  ψ  $\implies$  φ  $\preceq$  conn c l

```

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:

shows b : $FNot\ \varphi \preceq \psi \implies \varphi \preceq \psi$

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

using *subformula-into-subformula wf-conn-unary subformula-refl list.set-intros(1) subformula-refl*
by (*fastforce intro: subformula-into-subformula*)**+**

lemma *subformula-in-binary-conn*:

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

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: 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: wf-conn-list*)

lemma *subformula-not-incl-eq*:

assumes $\varphi \preceq \text{conn } c\ l$

and $\text{wf-conn } c\ l$

and $\forall \psi. \psi \in \text{set } l \longrightarrow \neg \varphi \preceq \psi$

shows $\varphi = \text{conn } c\ l$

using *assms* **apply** (*induction* $\text{conn } c\ l$ 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)
then have  $\varphi \preceq \text{conn } CAnd [\psi, \psi'] \longleftrightarrow$ 
  ( $\varphi = \text{conn } CAnd [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi'')$ )
  using wf-subformula-conn-cases by metis
then show ?P FAnd by auto
next
have wf-conn COr [ $\psi$ ,  $\psi'$ ] by (simp add: binary-connectives-def)
then have  $\varphi \preceq \text{conn } COr [\psi, \psi'] \longleftrightarrow$ 
  ( $\varphi = \text{conn } COr [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi'')$ )
  using wf-subformula-conn-cases by metis
then show ?P FOr by auto
next
have wf-conn CEq [ $\psi$ ,  $\psi'$ ] by (simp add: binary-connectives-def)
then have  $\varphi \preceq \text{conn } CEq [\psi, \psi'] \longleftrightarrow$ 
  ( $\varphi = \text{conn } CEq [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi'')$ )
  using wf-subformula-conn-cases by metis
then show ?P FEq by auto
next
have wf-conn CImp [ $\psi$ ,  $\psi'$ ] by (simp add: binary-connectives-def)
then have  $\varphi \preceq \text{conn } CImp [\psi, \psi'] \longleftrightarrow$ 
  ( $\varphi = \text{conn } CImp [\psi, \psi'] \vee (\exists \psi''. \psi'' \in \text{set } [\psi, \psi'] \wedge \varphi \preceq \psi'')$ )
  using wf-subformula-conn-cases by metis
then show ?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]:

```

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

```

proof (rule iffI)

```

{
  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 ?A

ultimately show ?B using wf by metis

next

assume ?B

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

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

case *nullary*
then have *False* **using** *corr* *incl* **by** *auto*
then show *vars-of-prop* $\psi \subseteq vars-of-prop\ (conn\ c\ l)$ **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*
then have $\psi = a \vee \psi = b$ **using** *incl* **by** *auto*
then show *vars-of-prop* $\psi \subseteq vars-of-prop\ (conn\ c\ l)$
using *ab c unfolding binary-connectives-def* **by** *auto*

next

case *unary* **note** *c = this*
fix $\varphi :: 'v\ propo$
have *l* = [ψ] **using** *corr c incl split-list* **by** *force*
then show *vars-of-prop* $\psi \subseteq vars-of-prop\ (conn\ c\ l)$ **using** *c* **by** *auto*

qed

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:

$\varphi \preceq \psi \implies vars-of-prop\ \varphi \subseteq vars-of-prop\ \psi$
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) = {} |
pos (FAnd  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FOr  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FEq  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FImp  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FNot  $\varphi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }

```

```

lemma finite-pos: finite (pos  $\varphi$ )
by (induct  $\varphi$ , auto)

```

```

lemma finite-inj-comp-set:

```

```

fixes s :: 'v set
assumes finite: finite s
and inj: inj f
shows card ({f p | p. p  $\in$  s}) = card s
using finite

```

```

proof (induct s rule: finite-induct)
show card {f p | p. p  $\in$  {}} = card {} by auto

```

```

next

```

```

fix x :: 'v and s :: 'v set
assume f: finite s and notin: x  $\notin$  s
and IH: card {f p | p. p  $\in$  s} = card s
have f': finite {f p | p. p  $\in$  insert x s} using f by auto
have notin': f x  $\notin$  {f p | p. p  $\in$  s} using notin inj injD by fastforce
have {f p | p. p  $\in$  insert x s} = insert (f x) {f p | p. p  $\in$  s} by auto
then have card {f p | p. p  $\in$  insert x s} = 1 + card {f p | p. p  $\in$  s}
using finite card-insert-disjoint f' notin' by auto
moreover have ... = card (insert x s) using notin f IH by auto
finally show card {f p | p. p  $\in$  insert x s} = card (insert x s) .

```

```

qed

```

```

lemma cons-inject:

```

```

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

```

```

lemma finite-insert-nil-cons:

```

```

finite s  $\implies$  card (insert [] {L # p | p. p  $\in$  s}) = 1 + card {L # p | p. p  $\in$  s}
using card-insert-disjoint by auto

```

```

lemma cord-not[simp]:

```

```

card (pos (FNot  $\varphi$ )) = 1 + card (pos  $\varphi$ )

```

```

by (simp add: cons-inject finite-inj-comp-set finite-pos)

```

```

lemma card-seperate:

```

```

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

then have $\dots \geq \text{card } (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2)$ **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* $:: \text{sign list} \Rightarrow 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **where**

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

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

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

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

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

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

lemma *path-to-subformula*:

path-to $p \ \varphi \ \varphi' \implies \varphi' \preceq \varphi$

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*

then show $\exists p. \text{path-to } p \ \varphi' \ \varphi'$ **by** *metis*

```

next
case (subformula-into-subformula  $\psi$   $l$   $c$ )
note  $wf = this(2)$  and  $IH = this(4)$  and  $\psi = this(1)$ 
then obtain  $p$  where  $p$ : path-to  $p$   $\psi$   $\varphi'$  by metis
{
  fix  $x :: 'v$ 
  assume  $c = CT \vee c = CF \vee c = CVar\ x$ 
  then have False using subformula-into-subformula by auto
  then have  $\exists p$ . path-to  $p$  (conn  $c$   $l$ )  $\varphi'$  by blast
}
moreover {
  assume  $c$ :  $c = CNot$ 
  then have  $l = [\psi]$  using wf  $\psi$  wf-conn-Not-decomp by fastforce
  then have path-to ( $L \# p$ ) (conn  $c$   $l$ )  $\varphi'$  by (metis  $c$  wf-conn-unary  $p$  path-to- $l$ )
  then have  $\exists p$ . path-to  $p$  (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  $c$  wf-conn-bin-list-length
    list-length2-decomp by metis
  then have  $a = \psi \vee b = \psi$  using  $\psi$  by auto
  then have path-to ( $L \# p$ ) (conn  $c$   $l$ )  $\varphi' \vee$  path-to ( $R \# p$ ) (conn  $c$   $l$ )  $\varphi'$  using  $c$  path-to- $l$ 
    path-to-r  $p$   $ab$  by (metis wf-conn-binary)
  then have  $\exists p$ . path-to  $p$  (conn  $c$   $l$ )  $\varphi'$  by blast
}
ultimately show  $\exists p$ . path-to  $p$  (conn  $c$   $l$ )  $\varphi'$  using connective-cases-arity by metis
qed

```

```

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

```

5 Semantics over the syntax

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

```

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

```


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

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

lemma *deduction-rule*:

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

proof

assume $H: \varphi \models_f \psi$
 $\{$
 fix A

“Suppose that 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 $A \models \varphi$
 then have $A \models \psi$ **using** H **unfolding** *evalf-def* **by** *metis*
 then have $A \models FImp \varphi \psi$ **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 $\neg A \models \varphi$
 then have $A \models FImp \varphi \psi$ **by** *auto*
 $\}$

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

ultimately have $A \models FImp \varphi \psi$ **by** *blast*
 $\}$
then show $\forall A. A \models FImp \varphi \psi$ **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*
 then have $\neg A \models FImp \varphi \psi$ **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 = this*
 then obtain a where $l: l = [a]$ **using** *wf-conn-Not-decomp wf* **by** *metis*
 obtain a' where $l': l' = [a']$ **using** *wf-conn-Not-decomp wf' c* **by** *metis*
 have $A \models a \longleftrightarrow A \models a'$ **using** $l l'$ **by** (*metis nth-Cons-0 same*)
 thus $A \models \text{conn } c l \longleftrightarrow A \models \text{conn } c l'$ **using** $l l' c$ **by** *auto*
next
 case *binary note c = this*
 then obtain $a b$ where $l: l = [a, b]$
using *wf-conn-bin-list-length list-length2-decomp wf* **by** *metis*
 obtain $a' b'$ where $l': l' = [a', b']$
using *wf-conn-bin-list-length list-length2-decomp wf' c* **by** *metis*

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

Relation between *propo-rew-step* and the rewriting we have seen before: *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 (*test-symb (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 *wf-conn c l*
 \Longrightarrow (*test-symb (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 (*test-symb (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 (*test-symb (FAnd $\varphi \psi$)* \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb (FOr $\varphi \psi$)*
 \longleftrightarrow (*test-symb (FOr $\varphi \psi$)* \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb (FNot φ)*
 \longleftrightarrow (*test-symb (FNot φ)* \wedge *all-subformula-st test-symb* φ)
and *all-subformula-st test-symb (FEq $\varphi \psi$)*
 \longleftrightarrow (*test-symb (FEq $\varphi \psi$)* \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)
and *all-subformula-st test-symb (FImp $\varphi \psi$)*
 \longleftrightarrow (*test-symb (FImp $\varphi \psi$)* \wedge *all-subformula-st test-symb* φ \wedge *all-subformula-st test-symb* ψ)

proof –

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

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

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

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

have $\text{all-subformula-st test-symb } (F\text{Not } \varphi) \longleftrightarrow \text{all-subformula-st test-symb } (\text{conn } C\text{Not } [\varphi])$
by *auto*
moreover have $\dots = (\text{test-symb } (\text{conn } C\text{Not } [\varphi]) \wedge (\forall \xi \in \text{set } [\varphi]. \text{all-subformula-st test-symb } \xi))$
using *all-subformula-st-decomp wf-conn-helper-facts(1)* **by** *metis*
finally show $\text{all-subformula-st test-symb } (F\text{Not } \varphi)$
 $\longleftrightarrow (\text{test-symb } (F\text{Not } \varphi) \wedge \text{all-subformula-st test-symb } \varphi)$ **by** *simp*
qed

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

lemma *subformula-all-subformula-st*:

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

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

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

}

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

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: elim-equiv.simps*)

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

ultimately show *?thesis*

using *no-test-symb-step-exists no-equiv test-symb-false-nullary unfolding no-equiv-def* **by** *blast*

qed

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

lemma *no-equiv-full-propo-rew-step-elim-equiv*:

$\text{full } (\text{propo-rew-step } \text{elim-equiv}) \ \varphi \ \psi \implies \text{no-equiv } \psi$

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 φ ψ) \longleftrightarrow (no-imp $\varphi \wedge$ no-imp ψ)
no-imp (FOr φ ψ) \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  $H': \forall \psi. \neg \text{elim-imp } FT \ \psi \ \forall \psi. \neg \text{elim-imp } FF \ \psi \ \forall \psi \ x. \neg \text{elim-imp } (FVar \ x) \ \psi$ 
    by (auto elim: elim-imp.cases)+
moreover
  have  $\bigwedge \varphi. \neg \text{no-imp-symb } \varphi \implies \exists \psi. \text{elim-imp } \varphi \ \psi$ 
    by (case-tac  $\varphi$ ) (force simp: elim-imp.simps)+
  then have  $(\bigwedge \varphi'. \varphi' \preceq \varphi \implies \neg \text{no-imp-symb } \varphi' \implies \exists \psi. \text{elim-imp } \varphi' \ \psi)$  by force
ultimately show ?thesis
  using no-test-symb-step-exists no-equiv test-symb-false-nullary unfolding no-imp-def by blast
qed

```

```

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

```

8.3 Eliminate all the True and False in the formula

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

inductive elimTB **where**

ElimTB1: elimTB (FAnd 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  $\varphi \ \psi:: 'b \text{ propo}$ 
  have  $\text{elimTB } \varphi \ \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$  by (induction rule: elimTB.inducts) auto
}
then show ?thesis using preserves-un-sat-def by auto
qed

```

inductive no-T-F-symb :: $'v \text{ propo} \Rightarrow \text{bool}$ **where**

no-T-F-symb-comp: $c \neq CF \implies c \neq CT \implies \text{wf-conn } c \ l \implies (\forall \varphi \in \text{set } l. \varphi \neq FT \wedge \varphi \neq FF)$
 $\implies \text{no-T-F-symb } (\text{conn } c \ l)$

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

$\text{wf-conn } c \ \psi s \implies$

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

unfolding no-T-F-symb.simps **apply** (cases c)

using wf-conn-list(1) **apply** fastforce

using wf-conn-list(2) **apply** fastforce

```

    using wf-conn-list(3) apply fastforce
    apply (metis (no-types, hide-lams) conn-inj connective.distinct(5,17))
    using conn-inj apply blast+
done

```

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

```

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

```

lemma *no-T-F-symb-false*[simp]:

```

fixes c :: 'v connective
shows
   $\neg$ 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  $\neg (\varphi = FT \vee \varphi = FF)$ 
  then have  $\forall \varphi' \in \text{set } [\varphi]. \varphi' \neq FT \wedge \varphi' \neq FF$  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))
  then show 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  $\varphi$ 

```


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

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* ($FNot\ \varphi$)
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 $c: c \in \text{binary-connectives}$
shows *no-T-F-symb-except-toplevel* ($conn\ c\ [\varphi, \psi]$)
by (*metis* (*no-types*, *lifting*) *assms* $c\ conn.simps(4)$ *list.discI* *noTrue-no-T-F-symb-except-toplevel*
wf-conn-no-T-F-symb-iff *no-T-F-symb-fnot* *set.ConsD* *wf-conn-binary* *wf-conn-helper-facts(1)*
wf-conn-list-decomp(1,2))

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:

fixes $l :: 'v\ propo\ list$ **and** $c :: 'v\ connective$
assumes *corr*: *wf-conn* $c\ l$
and $FT \in \text{set } l \vee FF \in \text{set } l$
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* ($FNot\ \varphi$)
by (*simp* *add*: *assms* *no-T-F-symb-except-toplevel.simps*)

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

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

no-T-F-except-top-level \equiv *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 $l :: 'v \text{ propo list}$ **and** $c :: 'v \text{ connective}$
assumes *wf-conn* $c \ l$
and $FT \in \text{set } l \vee FF \in \text{set } l$
shows $\neg \text{no-T-F-except-top-level } (\text{conn } c \ l)$
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 $\varphi \ \psi :: 'v \text{ propo}$
assumes $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
shows
 $\neg \text{no-T-F-except-top-level } (F\text{And } \varphi \ \psi)$
 $\neg \text{no-T-F-except-top-level } (F\text{Or } \varphi \ \psi)$
 $\neg \text{no-T-F-except-top-level } (F\text{Eq } \varphi \ \psi)$
 $\neg \text{no-T-F-except-top-level } (F\text{Imp } \varphi \ \psi)$
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*:
 $\text{no-T-F-symb-except-toplevel } \varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies \text{no-T-F-symb } \varphi$
by (*induct rule: no-T-F-symb-except-toplevel.induct, auto*)

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

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

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

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

proof –

have *wf: wf-conn* $c \ [\varphi, \psi]$ **using** c **by** *auto*
then have $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F-symb } (\text{conn } c \ [\varphi, \psi]) \wedge \text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
by (*simp add: all-subformula-st-decomp no-T-F-def*)
then show $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
using c *wf all-subformula-st-decomp list.discI no-T-F-def no-T-F-symb-except-toplevel-bin-decom*
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 no-T-F (conn c [ $\varphi$ ,  $\psi$ ])  $\longleftrightarrow$  (no-T-F  $\varphi \wedge$  no-T-F  $\psi$ )
  using no-T-F-bin-decomp assms unfolding binary-connectives-def by blast

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

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

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

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

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

```

```

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

```

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

```

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

```

lemma *elimTB-inv*:

```

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

```

```

}
then show no-imp  $\psi$ 
  using full-propo-rew-step-inv-stay-conn[of elimTB no-imp-symb  $\varphi$   $\psi$ ] assms
  no-imp-symb-conn-characterization unfolding no-imp-def by metis
qed

```

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

8.4 PushNeg

Push the negation inside the formula, until the litteral.

inductive *pushNeg* **where**

```

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

```

lemma *pushNeg-transformation-consistent*:

```

 $A \models \text{FNot (FAnd } \varphi \psi) \longleftrightarrow A \models (\text{FOr (FNot } \varphi) (\text{FNot } \psi))$ 
 $A \models \text{FNot (FOr } \varphi \psi) \longleftrightarrow A \models (\text{FAnd (FNot } \varphi) (\text{FNot } \psi))$ 
 $A \models \text{FNot (FNot } \varphi) \longleftrightarrow A \models \varphi$ 
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 *pushNeg-consistent*: preserves-un-sat pushNeg
 unfolding preserves-un-sat-def by (simp add: pushNeg-explicit)

lemma *pushNeg-lifted-consistant*:

```

preserves-un-sat (full (propo-rew-step pushNeg))
by (simp add: pushNeg-consistent)

```

fun *simple* **where**

```

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

```

lemma *simple-decomp*:

```

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

```

lemma *subformula-conn-decomp-simple*:

```

fixes  $\varphi \psi :: 'v$  propo
assumes s: simple  $\psi$ 
shows  $\varphi \preceq \text{FNot } \psi \longleftrightarrow (\varphi = \text{FNot } \psi \vee \varphi = \psi)$ 

```

proof –

```

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

```

using *subformula-conn-decomp wf-conn-helper-facts(1)* **by** *metis*
then show $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \vee \varphi = \psi)$ **using** *s* **by** *(auto simp: 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: 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* $\psi \implies \exists \psi'. \text{pushNeg}\ \psi\ \psi'$
apply (*induct* ψ , *auto*)
apply (*rename-tac* ψ , *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* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists \psi\ \psi'. \psi \preceq \varphi \wedge \text{pushNeg}\ \psi\ \psi'$
proof -
have $\forall x. \text{simple-not-symb}\ (FF :: 'v\ propo) \wedge \text{simple-not-symb}\ FT \wedge \text{simple-not-symb}\ (FVar\ (x :: 'v))$
by *auto*
moreover {
fix $c :: 'v\ connective$ **and** $l :: 'v\ propo\ list$ **and** $\psi :: 'v\ propo$
have $H: \text{pushNeg}\ (\text{conn}\ c\ l)\ \psi \implies \neg \text{simple-not-symb}\ (\text{conn}\ c\ l)$
by (*cases* (*conn* $c\ l$) *rule: pushNeg.cases*) *auto*
}
moreover {
fix $x :: 'v$
have $H': \text{simple-not}\ FT\ \text{simple-not}\ FF\ \text{simple-not}\ (FVar\ x)$
by *simp-all*
}
moreover {
fix $\psi :: 'v\ propo$
have $\psi \preceq \varphi \implies \neg$ *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:  $c \neq CF \wedge c \neq CT \wedge \varphi \neq FF \wedge \varphi \neq FT \wedge (\forall \psi \in \text{set } \xi \cup \text{set } \xi'. \psi \neq FF \wedge \psi \neq FT)$ 
  then have  $c \neq CF \wedge c \neq CT \wedge \text{wf-conn } c$  ( $\xi @ \varphi' \# \xi'$ )
    using wf-conn-no-arity-change-helper wf-conn-no-arity-change by metis
  moreover have  $n'$ : 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 then have 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
    then have  $\forall \psi \in \text{set } (\xi @ \varphi' \# \xi'). \psi \neq FF \wedge \psi \neq FT$  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
then show ?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 (prop-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  $c$  ( $\xi$  @  $\varphi'$  #  $\xi'$ ))
  using all-subformula-st-test-symb-true-phi
  by (fastforce simp: no-T-F-def all-subformula-st-decomp wf wf')
qed

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

```

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

```

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

```

8.5 Push inside

inductive *push-conn-inside* :: 'v connective \Rightarrow 'v connective \Rightarrow 'v propo \Rightarrow 'v propo \Rightarrow bool
for *c c'* :: 'v connective **where**
push-conn-inside-l[simp]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
 $\Longrightarrow \text{push-conn-inside } c \ c' \ (\text{conn } c \ [\text{conn } c' \ [\varphi 1, \varphi 2], \psi])$
 $(\text{conn } c' \ [\text{conn } c \ [\varphi 1, \psi], \text{conn } c \ [\varphi 2, \psi]]) \mid$
push-conn-inside-r[simp]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
 $\Longrightarrow \text{push-conn-inside } c \ c' \ (\text{conn } c \ [\psi, \text{conn } c' \ [\varphi 1, \varphi 2]])$
 $(\text{conn } c' \ [\text{conn } c \ [\psi, \varphi 1], \text{conn } c \ [\psi, \varphi 2]])$

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

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

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

inductive *not-c-in-c'-symb* :: 'v connective \Rightarrow 'v connective \Rightarrow 'v propo \Rightarrow bool **for** *c c'* **where**
not-c-in-c'-symb-l[simp]: $\text{wf-conn } c \ [\text{conn } c' \ [\varphi, \varphi'], \psi] \Longrightarrow \text{wf-conn } c' \ [\varphi, \varphi']$
 $\Longrightarrow \text{not-c-in-c'-symb } c \ c' \ (\text{conn } c \ [\text{conn } c' \ [\varphi, \varphi'], \psi]) \mid$
not-c-in-c'-symb-r[simp]: $\text{wf-conn } c \ [\psi, \text{conn } c' \ [\varphi, \varphi']] \Longrightarrow \text{wf-conn } c' \ [\varphi, \varphi']$
 $\Longrightarrow \text{not-c-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \text{conn } c' \ [\varphi, \varphi']])$

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

lemma *c-in-c'-symb-simp*:
 $\text{not-c-in-c'-symb } c \ c' \ \xi \Longrightarrow \xi = FF \vee \xi = FT \vee \xi = FVar \ x \vee \xi = FNot \ FF \vee \xi = FNot \ FT$
 $\vee \xi = FNot \ (FVar \ x) \Longrightarrow \text{False}$
apply (induct rule: *not-c-in-c'-symb.induct*, auto *simp*: *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]*:
 $\neg \text{not-c-in-c'-symb } c \ c' \ FF$
 $\neg \text{not-c-in-c'-symb } c \ c' \ FT$

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

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

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

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

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

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

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

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

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

next

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

qed

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

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

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

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

proof –

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

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

lemma $\text{not-}c\text{-in-}c'\text{-simp}[\text{simp}]$:
fixes $\varphi1 \ \varphi2 \ \psi :: 'v \text{ propo}$ **and** $x :: 'v$
shows
 $c\text{-in-}c'\text{-symb } c \ c' \ FT$
 $c\text{-in-}c'\text{-symb } c \ c' \ FF$
 $c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$
 $\text{wf-conn } c \ [\text{conn } c' \ [\varphi1, \varphi2], \psi] \implies \text{wf-conn } c' \ [\varphi1, \varphi2]$
 $\implies \neg c\text{-in-}c'\text{-only } c \ c' \ (\text{conn } c \ [\text{conn } c' \ [\varphi1, \varphi2], \psi])$
apply ($\text{simp-all add: } c\text{-in-}c'\text{-only-def}$)
using $\text{all-subformula-st-test-symb-true-phi not-}c\text{-in-}c'\text{-symb-l}$ **by** blast

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

lemma $c\text{-in-}c'\text{-symb-step-exists}$:
fixes $\varphi :: 'v \text{ propo}$
assumes $c: c = CAnd \vee c = COr$ **and** $c': c' = CAnd \vee c' = COr$
shows $\psi \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi \implies \exists \psi'. \text{push-conn-inside } c \ c' \ \psi \ \psi'$
apply ($\text{induct } \psi \text{ rule: propo-induct-arity}$)
apply $\text{auto}[2]$
proof –
fix $\psi1 \ \psi2 \ \varphi' :: 'v \text{ propo}$
assume $IH\psi1: \psi1 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi1 \implies \text{Ex } (\text{push-conn-inside } c \ c' \ \psi1)$
and $IH\psi2: \psi2 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi2 \implies \text{Ex } (\text{push-conn-inside } c \ c' \ \psi2)$
and $\varphi': \varphi' = FAnd \ \psi1 \ \psi2 \vee \varphi' = FOr \ \psi1 \ \psi2 \vee \varphi' = FImp \ \psi1 \ \psi2 \vee \varphi' = FEq \ \psi1 \ \psi2$
and $\text{in}\varphi: \varphi' \preceq \varphi$ **and** $n0: \neg c\text{-in-}c'\text{-symb } c \ c' \ \varphi'$
then have $n: \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ \varphi'$ **by** auto
{
assume $\varphi': \varphi' = \text{conn } c \ [\psi1, \psi2]$
obtain $a \ b$ **where** $\psi1 = \text{conn } c' \ [a, b] \vee \psi2 = \text{conn } c' \ [a, b]$
using $n \ \varphi'$ **apply** ($\text{induct rule: not-}c\text{-in-}c'\text{-symb.induct}$)
using c **by** force+
then have $\text{Ex } (\text{push-conn-inside } c \ c' \ \varphi')$
unfolding φ' **apply** auto
using $\text{push-conn-inside.intros}(1) \ c \ c'$ **apply** blast
using $\text{push-conn-inside.intros}(2) \ c \ c'$ **by** blast
}
moreover {

```

  assume  $\varphi'$ :  $\varphi' \neq \text{conn } c [\psi 1, \psi 2]$ 
  have  $\forall \varphi \ c \ ca. \exists \varphi 1 \ \psi 1 \ \psi 2 \ \psi 1' \ \psi 2' \ \varphi 2'. \text{conn } (c::'v \text{ connective}) [\varphi 1, \text{conn } ca [\psi 1, \psi 2]] = \varphi$ 
     $\vee \text{conn } c [\text{conn } ca [\psi 1', \psi 2'], \varphi 2'] = \varphi \vee c\text{-in-}c'\text{-symb } c \ ca \ \varphi$ 
  by (metis not-c-in-c'-symb.cases)
  then have  $Ex \ (\text{push-conn-inside } c \ c' \ \varphi')$ 
  by (metis (no-types)  $c \ c' \ n \ \text{push-conn-inside-l} \ \text{push-conn-inside-r}$ )
}
ultimately show  $Ex \ (\text{push-conn-inside } c \ c' \ \varphi')$  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: assms(2)  $c' \ c\text{-in-}c'\text{-symb-step-exists}$ )
  }
  ultimately show ?thesis using noTB no-test-symb-step-exists[of  $c\text{-in-}c'\text{-symb } c \ c'$ ]
  unfolding c-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 ( $\text{push-conn-inside } c \ c' \ \varphi \ \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$ )
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi \ \psi$ )
  then show  $\text{no-T-F } \psi$ 
  by (cases rule: push-conn-inside.cases, auto)
next
  case (propo-rew-one-step-lift  $\varphi \ \varphi' \ c \ \xi \ \xi'$ )
  note  $\text{rel} = \text{this}(1)$  and  $IH = \text{this}(2)$  and  $\text{wf} = \text{this}(3)$  and  $\text{no-T-F} = \text{this}(4)$ 
  have  $\text{no-T-F } \varphi$ 
  using  $\text{wf no-T-F no-T-F-def subformula-into-subformula subformula-all-subformula-st}$ 
     $\text{subformula-refl}$  by (metis (no-types) in-set-conv-decomp)
  then have  $\varphi': \text{no-T-F } \varphi'$  using  $IH$  by blast

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

```

```

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

```

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

lemma *simple-propo-rew-step-inv-push-conn-inside-simple-not*:
fixes *c c' :: 'v* **connective** **and** *φ ψ :: 'v* **propo**
shows *propo-rew-step (push-conn-inside c c') φ ψ ⇒ simple-not φ ⇒ simple-not ψ*
proof (induct rule: *propo-rew-step.induct*)
case (*global-rel φ ψ*)
then show ?*case* **by** (cases φ, auto simp: *push-conn-inside.simps*)
next
case (*propo-rew-one-step-lift φ φ' ca ξ ξ'*) **note** *rew = this(1)* **and** *IH = this(2)* **and** *wf = this(3)*
and *simple = this(4)*
show ?*case*
proof (cases *ca* rule: *connective-cases-arity*)
case *nullary*
then show ?*thesis* **using** *propo-rew-one-step-lift* **by** auto
next
case *binary* **note** *ca = this*
obtain *a b* **where** *ab: ξ @ φ' # ξ' = [a, b]*
using *wf ca list-length2-decomp wf-conn-bin-list-length*
by (metis (*no-types*) *wf-conn-no-arity-change-helper*)
have $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{simple-not } \zeta$
by (metis *wf all-subformula-st-decomp simple simple-not-def*)
then have $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \text{simple-not } \zeta$ **using** *IH* **by** *simp*
moreover have *simple-not-symb (conn ca (ξ @ φ' # ξ'))* **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 ?thesis
    by (simp add: ab all-subformula-st-decomp ca)
next
  case unary
  then show ?thesis
    using rew simple-propo-rew-step-push-conn-inside-inv[OF rew] IH local.wf simple by auto
qed
qed

```

```

lemma propo-rew-step-push-conn-inside-simple-not:
  fixes  $\varphi \varphi' :: 'v \text{ propo}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $c :: 'v \text{ connective}$ 
  assumes
    propo-rew-step (push-conn-inside  $c \ c'$ )  $\varphi \varphi'$  and
    wf-conn  $c \ (\xi @ \varphi \# \xi')$  and
    simple-not-symb (conn  $c \ (\xi @ \varphi \# \xi')$ ) and
    simple-not-symb  $\varphi'$ 
  shows simple-not-symb (conn  $c \ (\xi @ \varphi' \# \xi')$ )
  using assms
proof (induction rule: propo-rew-step.induct)
print-cases
  case (global-rel)
  then show ?case
    by (metis conn.simps(12,17) list.discI push-conn-inside.cases simple-not-symb.elims(3)
      wf-conn-helper-facts(5) wf-conn-list(2) wf-conn-list(8) wf-conn-no-arity-change
      wf-conn-no-arity-change-helper)
next
  case (propo-rew-one-step-lift  $\varphi \varphi' c' \chi s \chi s'$ ) note tel = this(1) and wf = this(2) and
    IH = this(3) and wf' = this(4) and simple' = this(5) and simple = this(6)
  then show ?case
    proof (cases  $c'$  rule: connective-cases-arity)
    case nullary
    then show ?thesis using wf simple simple' by auto
    next
    case binary note  $c = \text{this}(1)$ 
    have corr': wf-conn  $c \ (\xi @ \text{conn } c' (\chi s @ \varphi' \# \chi s') \# \xi')$ 
      using wf wf-conn-no-arity-change
      by (metis wf' wf-conn-no-arity-change-helper)
    then show ?thesis
      using  $c \text{ propo-rew-one-step-lift } wf$ 
      by (metis conn.simps(17) connective.distinct(37) propo-rew-step-subformula-imp
        push-conn-inside.cases simple-not-symb.elims(3) wf-conn.simps wf-conn-list(2,8))
    next
    case unary
    then have empty:  $\chi s = [] \ \chi s' = []$  using wf by auto
    then show ?thesis using simple unary simple' wf wf'
      by (metis connective.distinct(37) connective.distinct(39) propo-rew-step-subformula-imp
        push-conn-inside.cases simple-not-symb.elims(3) tel wf-conn-list(8)
        wf-conn-no-arity-change wf-conn-no-arity-change-helper)
    qed
  qed
qed

```

```

lemma push-conn-inside-not-true-false:
  push-conn-inside  $c \ c' \varphi \psi \implies \psi \neq FT \wedge \psi \neq FF$ 

```

by (induct rule: push-conn-inside.induct, auto)

lemma push-conn-inside-inv:

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

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

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

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

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

using H apply simp

proof (rename-tac $\varphi \ \varphi'$ ca $\psi s \ \psi s'$, case-tac ca rule: connective-cases-arity)

fix $\varphi \ \varphi' :: 'v \text{ propo}$ and $c :: 'v \text{ connective}$ and $\xi \ \xi' :: 'v \text{ propo list}$

and $x :: 'v$

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

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

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

then have False by auto

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

next

fix $\varphi \ \varphi' :: 'v \text{ propo}$ and $ca :: 'v \text{ connective}$ 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

then have simple-not:all-subformula-st simple-not-symb (FNot φ) using corr $c \ n$ by auto

then have simple φ

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

then have simple φ'

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

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

by (metis simple-not $\varphi\text{-}\varphi'$ append-Nil conn.simps(4) all-subformula-st-decomp-explicit(3) simple-not-symb.simps(1))

next

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

and $x :: 'v$

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

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

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

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

and c : $ca \in \text{binary-connectives}$


```

have all-subformula-st simple-not-symb  $\varphi$ 
  using  $n$   $c$  corr all-subformula-st-decomp by fastforce
then have  $\varphi'$ : 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
then have  $\xi @ \varphi' \# \xi' = [a, \varphi'] \vee (\xi @ \varphi' \# \xi') = [\varphi', b]$ 
  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
  then have all-subformula-st simple-not-symb  $\chi \vee \chi \notin \text{set } (\xi @ \varphi' \# \xi')$ 
    using  $wf'$  by (metis (no-types)  $\varphi'$  all-subformula-st-decomp calculation insert-iff
      list.set(2))
}
then have  $\forall \varphi. \varphi \in \text{set } (\xi @ \varphi' \# \xi') \longrightarrow \text{all-subformula-st simple-not-symb } \varphi$ 
  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  $c$   $c'$ )  $\varphi \varphi' \Longrightarrow wf\text{-conn } ca (\xi @ \varphi \# \xi')$ 
     $\Longrightarrow \text{simple-not-symb } (\text{conn } ca (\xi @ \varphi \# \xi')) \Longrightarrow \text{simple-not-symb } \varphi'$ 
     $\Longrightarrow \text{simple-not-symb } (\text{conn } ca (\xi @ \varphi' \# \xi'))$ 
  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  $c$   $c'$ )  $\varphi \psi \Longrightarrow \text{no-T-F-except-top-level } \varphi$ 
     $\Longrightarrow \text{no-T-F-except-top-level } \psi$ 
  proof –
    assume rel: propo-rew-step (push-conn-inside  $c$   $c'$ )  $\varphi \psi$ 
    and no-T-F-except-top-level  $\varphi$ 
    then have  $\text{no-T-F } \varphi \vee \varphi = FF \vee \varphi = FT$ 
      by (metis no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
    moreover {

```

```

    assume  $\varphi = FF \vee \varphi = FT$ 
    then have False using rel propo-rew-step-push-conn-inside by blast
    then have no-T-F-except-top-level  $\psi$  by blast
  }
  moreover {
    assume no-T-F  $\varphi \wedge \varphi \neq FF \wedge \varphi \neq FT$ 
    then have no-T-F  $\psi$  using rel push-conn-insidec-in-c'-symb-no-T-F by blast
    then have 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 ( $\xi @ \varphi \# \xi'$ )
  then have 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
    then have  $\varphi \neq FT \wedge \varphi \neq FF$  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)
    then have  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \zeta \neq FT \wedge \zeta \neq FF$  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  $\varphi \psi :: 'v$  propo
  have H: 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)
}
then show 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  $\varphi \psi :: 'v$  propo
  have H: 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)
}

```

then show *no-imp* ψ
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 $\varphi \psi :: 'v$ *propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step* (*push-conn-inside* c c')) $\varphi \psi$ **and**
no-T-F-except-top-level φ **and**
simple-not φ **and**
 $c = CAnd \vee c = COr$ **and**
 $c' = CAnd \vee c' = COr$
shows *c-in-c'-only* c $c' \psi$
using *c-in-c'-symb-rew* *assms* *full-propo-rew-step-subformula* **by** *blast*

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

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

lemma *only-c-inside-symb-simp*[*simp*]:
only-c-inside-symb c *FF* *only-c-inside-symb* c *FT* *only-c-inside-symb* c (*FVar* x) **by** *auto*

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

lemma *only-c-inside-symb-decomp*:
only-c-inside-symb c $\psi \longleftrightarrow$ (*simple* ψ
 $\vee (\exists \varphi'. \psi = FNot \varphi' \wedge$ *simple* $\varphi')$
 $\vee (\exists l. \psi =$ *conn* c $l \wedge$ *wf-conn* c l))
by (*auto simp: 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* ψ) \longleftrightarrow *simple* ψ
apply (*auto simp: only-c-inside-symb.intros(3)*)
by (*induct FNot* ψ *rule: only-c-inside-symb.induct, auto simp: wf-conn-list(8) c*)

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

$(\forall \psi. \psi \preceq \varphi \longrightarrow (\text{simple } \psi \vee (\exists \varphi'. \psi = \text{FNot } \varphi' \wedge \text{simple } \varphi') \vee (\exists l. \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l)))$

unfolding *only-c-inside-def* **by** (*auto simp: all-subformula-st-def only-c-inside-symb-decomp*)

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

fixes $c \ c' :: 'v \text{ connective}$ **and** $l :: 'v \text{ propo list}$ **and** $\varphi :: 'v \text{ propo}$
assumes $cc': c \neq c'$ **and** $c: c = \text{CAnd} \vee c = \text{COr}$ **and** $c': c' = \text{CAnd} \vee c' = \text{COr}$
and *only*: *only-c-inside* $c \ \varphi$ **and** *incl*: $\text{conn } c' \ l \preceq \varphi$ **and** *wf*: $\text{wf-conn } c' \ l$
shows *False*

proof –

let $? \psi = \text{conn } c' \ l$
have $\text{simple } ? \psi \vee (\exists \varphi'. ? \psi = \text{FNot } \varphi' \wedge \text{simple } \varphi') \vee (\exists l. ? \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l)$
using *only-c-inside-decomp only incl* **by** *blast*
moreover **have** $\neg \text{simple } ? \psi$
using *wf simple-decomp* **by** (*metis c' connective.distinct(19) connective.distinct(7,9,21,29,31)*
wf-conn-list(1-3))

moreover

{
fix φ'
have $? \psi \neq \text{FNot } \varphi'$ **using** $c' \text{ conn-inj-not}(1)$ *wf* **by** *blast*
}

ultimately obtain $l :: 'v \text{ propo list}$ **where** $? \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l$ **by** *metis*

then **have** $c = c'$ **using** *conn-inj wf* **by** *metis*

then **show** *False* **using** cc' **by** *auto*

qed

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

assumes $\delta: c \neq c'$ **and** $c: c = \text{CAnd} \vee c = \text{COr}$ **and** $c': c' = \text{CAnd} \vee c' = \text{COr}$
shows *only-c-inside* $c \ \varphi \implies \text{c-in-c'-symb } c \ c' \ \varphi$
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 $l :: 'v \text{ propo list}$ **and** $c \ c' \ ca :: 'v \text{ connective}$
shows $\text{wf-conn } ca \ l \implies ca \neq c \implies \text{c-in-c'-symb } c \ c' \ (\text{conn } ca \ l)$

proof –

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

qed

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

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

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

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

```

  shows wf-conn c l  $\implies$  c-in-c'-only c c' (conn c l)  $\implies$  ( $\forall \psi \in \text{set } l. \text{ only-c-inside } c \psi$ )
using inv
proof (induct conn c l arbitrary: l rule: propo-induct-arity)
  case (nullary x)
  then show ?case by (auto simp: wf-conn-list assms)
next
  case (unary  $\varphi$  la)
  then have c = CNot  $\wedge$  la = [ $\varphi$ ] by (metis (no-types) wf-conn-list(8))
  then show ?case using assms(2) assms(1) by blast
next
  case (binary  $\varphi_1$   $\varphi_2$ )
  note IH $\varphi_1$  = this(1) and IH $\varphi_2$  = this(2) and  $\varphi$  = this(3) and only = this(5) and wf = this(4)
  and no-equiv = this(6) and no-imp = this(7) and simple-not = this(8)
  then have l: l = [ $\varphi_1$ ,  $\varphi_2$ ] by (meson wf-conn-list(4-7))
  let ? $\varphi$  = conn c l

  obtain c1 l1 c2 l2 where  $\varphi_1$ :  $\varphi_1$  = conn c1 l1 and wf $\varphi_1$ : wf-conn c1 l1
  and  $\varphi_2$ :  $\varphi_2$  = conn c2 l2 and wf $\varphi_2$ : wf-conn c2 l2 using exists-c-conn by metis
  then have c-in-only $\varphi_1$ : c-in-c'-only c c' (conn c1 l1) and c-in-c'-only c c' (conn c2 l2)
  using only l unfolding c-in-c'-only-def using assms(1) by auto
  have inc $\varphi_1$ :  $\varphi_1 \preceq ?\varphi$  and inc $\varphi_2$ :  $\varphi_2 \preceq ?\varphi$ 
  using  $\varphi_1$   $\varphi_2$   $\varphi$  local.wf by (metis conn.simps(5-8) helper-fact subformula-in-binary-conn(1,2))+

  have c1-eq: c1  $\neq$  CEq and c2-eq: c2  $\neq$  CEq
  unfolding no-equiv-def using inc $\varphi_1$  inc $\varphi_2$  by (metis  $\varphi_1$   $\varphi_2$  wf $\varphi_1$  wf $\varphi_2$  assms(1) no-equiv
  no-equiv-eq(1) no-equiv-symb.elims(3) no-equiv-symb-conn-characterization wf-conn-list(4,5)
  no-equiv-def subformula-all-subformula-st)+
  have c1-imp: c1  $\neq$  CImp and c2-imp: c2  $\neq$  CImp
  using no-imp by (metis  $\varphi_1$   $\varphi_2$  all-subformula-st-decomp-explicit-imp(2,3) assms(1)
  conn.simps(5,6) l no-imp-imp(1) no-imp-symb.elims(3) no-imp-symb-conn-characterization
  wf $\varphi_1$  wf $\varphi_2$  all-subformula-st-decomp no-imp-symb-conn-characterization)+
  have c1c: c1  $\neq$  c'
  proof
    assume c1c: c1 = c'
    then obtain  $\xi_1$   $\xi_2$  where l1: l1 = [ $\xi_1$ ,  $\xi_2$ ]
    by (metis assms(2) connective.distinct(37,39) helper-fact wf $\varphi_1$  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
  then have ( $\varphi_1$  = conn c l1  $\wedge$  wf-conn c l1)  $\vee$  ( $\exists \psi_1. \varphi_1$  = 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  $\varphi_1$  = conn c l1  $\wedge$  wf-conn c l1
    then have only-c-inside c  $\varphi_1$ 
    by (metis IH $\varphi_1$   $\varphi_1$  all-subformula-st-decomp-imp inc $\varphi_1$  no-equiv no-equiv-def no-imp no-imp-def
    c-in-only $\varphi_1$  only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
    subformula-all-subformula-st)
  }
  moreover {
    assume  $\exists \psi_1. \varphi_1$  = FNot  $\psi_1$ 
    then obtain  $\psi_1$  where  $\varphi_1$  = FNot  $\psi_1$  by metis
    then have only-c-inside c  $\varphi_1$ 

```

```

    by (metis all-subformula-st-def assms(1) connective.distinct(37,39) incφ1
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
  }
  moreover {
    assume simple φ1
    then have only-c-inside c φ1
      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-insideφ1: only-c-inside c φ1 by metis

  have c-in-onlyφ2: c-in-c'-only c c' (conn c2 l2)
    using only l φ2 wfφ2 assms unfolding c-in-c'-only-def by auto
  have c2c: c2 ≠ c'
  proof
    assume c2c: c2 = c'
    then obtain ξ1 ξ2 where l2: l2 = [ξ1, ξ2]
      by (metis assms(2) wfφ2 wf-conn.simps connective.distinct(7,9,19,21,29,31,37,39))
    then have c-in-c'-symb c c' (conn c [φ1, conn c' l2])
      using c2c l only φ2 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 [φ1, conn c' l2])
      using assms(1) c2c l2 not-c-in-c'-symb-r wfφ2 wf-conn-helper-facts(5,6) by metis
    ultimately show False by auto
  qed
  then have (φ2 = conn c l2 ∧ wf-conn c l2) ∨ (∃ψ2. φ2 = FNot ψ2) ∨ simple φ2
    using c2-eq by (metis φ2 assms(1-3) c2-eq c2-imp simple.elims(3) wfφ2 wf-conn-list(4-7))
  moreover {
    assume φ2 = conn c l2 ∧ wf-conn c l2
    then have only-c-inside c φ2
      by (metis IHφ2 φ2 all-subformula-st-decomp incφ2 no-equiv no-equiv-def no-imp no-imp-def
          c-in-onlyφ2 only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
          subformula-all-subformula-st)
  }
  moreover {
    assume ∃ψ2. φ2 = FNot ψ2
    then obtain ψ2 where φ2 = FNot ψ2 by metis
    then have only-c-inside c φ2
      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
    then have 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*) φ ψ **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 [φ , ψ]
 \Rightarrow *grouped-by* c (*conn* c [φ , ψ])

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

lemma *only-c-inside-symb-c-eq-c'*:
only-c-inside-symb c (*conn* c' [$\varphi 1$, $\varphi 2$]) \Rightarrow $c' = CAnd \vee c' = COr \Rightarrow$ *wf-conn* c' [$\varphi 1$, $\varphi 2$]
 \Rightarrow $c' = c$
by (*induct* *conn* c' [$\varphi 1$, $\varphi 2$] *rule*: *only-c-inside-symb.induct*, *auto* *simp*: *conn-inj*)

lemma *only-c-inside-c-eq-c'*:
only-c-inside c (*conn* c' [$\varphi 1$, $\varphi 2$]) \Rightarrow $c' = CAnd \vee c' = COr \Rightarrow$ *wf-conn* c' [$\varphi 1$, $\varphi 2$] \Rightarrow $c = c'$
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 $c: c \neq CNot$ **and** $c': c' = CAnd \vee c' = COr$
shows *only-c-inside* c $\varphi \Rightarrow$ *grouped-by* c φ (**is** ?*O* $\varphi \Rightarrow$?*G* φ)
proof (*induct* φ *rule*: *propo-induct-arity*)
case (*nullary* φ x)
then show ?*G* φ **by** *auto*
next
case (*unary* ψ)
then show ?*G* (*FNot* ψ) **by** (*auto* *simp*: 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 $\varphi\text{-conn}$: $\varphi = \text{conn } c$ [$\varphi 1$, $\varphi 2$] **and** *wf*: *wf-conn* c [$\varphi 1$, $\varphi 2$]


```

proof –
  obtain  $c''\ l''$  where  $\varphi\text{-}c'': \varphi = \text{conn } c''\ l''$  and  $\text{wf}: \text{wf-conn } c''\ l''$ 
    using exists-c-conn by metis
  then have  $l'': l'' = [\varphi 1, \varphi 2]$  using  $\varphi$  by (metis wf-conn-list(4–7))
  have only-c-inside-symb  $c$  ( $\text{conn } c''\ [\varphi 1, \varphi 2]$ )
    using only all-subformula-st-test-symb-true-phi
    unfolding only-c-inside-def  $\varphi\text{-}c''\ l''$  by metis
  then have  $c = c''$ 
    by (metis  $\varphi\ \varphi\text{-}c''\ \text{conn-inj}\ \text{conn-inj-not}(2)\ l''\ \text{list.distinct}(1)\ \text{list.inject}\ \text{wf}$ 
      only-c-inside-symb.cases simple.simps(5–8))
  then show  $\varphi = \text{conn } c\ [\varphi 1, \varphi 2]$  and  $\text{wf-conn } c\ [\varphi 1, \varphi 2]$  using  $\varphi\text{-}c''\ \text{wf } l''$  by auto
qed
have grouped-by  $c\ \varphi 1$  using  $\text{wf } IH\ \varphi 1\ IH\ \varphi 2\ \varphi\text{-conn}\ \text{only } \varphi$  unfolding only-c-inside-def by auto
moreover have grouped-by  $c\ \varphi 2$ 
  using  $\text{wf } \varphi\ IH\ \varphi 1\ IH\ \varphi 2\ \varphi\text{-conn}\ \text{only}$  unfolding only-c-inside-def by auto
ultimately show  $?G\ \varphi$  using  $\varphi\text{-conn}\ \text{connected-is-group}\ \text{local.wf}$  by blast
qed

```

lemma *grouped-by-false*:

```

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

```

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

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

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

```

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

```

lemma *c-in-c'-only-super-grouped-by*:

```

assumes  $c: c = CAnd \vee c = COr$  and  $c': c' = CAnd \vee c' = COr$  and  $cc': c \neq c'$ 
shows  $\text{no-equiv } \varphi \implies \text{no-imp } \varphi \implies \text{simple-not } \varphi \implies \text{c-in-c'-only } c\ c'\ \varphi$ 
 $\implies \text{super-grouped-by } c\ c'\ \varphi$ 
(is  $?NE\ \varphi \implies ?NI\ \varphi \implies ?SN\ \varphi \implies ?C\ \varphi \implies ?S\ \varphi)$ 

```

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

```

case (nullary  $\varphi\ x$ )
then show  $?S\ \varphi$  by auto

```

next

```

case (unary  $\varphi$ )
then have simple-not-symb ( $FNot\ \varphi$ )
  using all-subformula-st-test-symb-true-phi unfolding simple-not-def by blast
then have  $\varphi = FT \vee \varphi = FF \vee (\exists\ x. \varphi = FVar\ x)$  by (cases  $\varphi$ , auto)
then show  $?S\ (FNot\ \varphi)$  by auto

```

```

next
case (binary  $\varphi$   $\varphi 1$   $\varphi 2$ )
note  $IH\varphi 1 = \text{this}(1)$  and  $IH\varphi 2 = \text{this}(2)$  and  $\text{no-equiv} = \text{this}(4)$  and  $\text{no-imp} = \text{this}(5)$ 
and  $\text{simpleN} = \text{this}(6)$  and  $\text{c-in-c'-only} = \text{this}(7)$  and  $\varphi' = \text{this}(3)$ 
{
  assume  $\varphi = FImp\ \varphi 1\ \varphi 2 \vee \varphi = FEq\ \varphi 1\ \varphi 2$ 
  then have False using no-equiv no-imp by auto
  then have  $?S\ \varphi$  by auto
}
moreover {
  assume  $\varphi: \varphi = \text{conn}\ c'\ [\varphi 1, \varphi 2] \wedge \text{wf-conn}\ c'\ [\varphi 1, \varphi 2]$ 
  have c-in-c'-only:  $\text{c-in-c'-only}\ c'\ \varphi 1 \wedge \text{c-in-c'-only}\ c'\ \varphi 2 \wedge \text{c-in-c'-symb}\ c'\ c'\ \varphi$ 
  using c-in-c'-only  $\varphi'$  unfolding c-in-c'-only-def by auto
  have super-grouped-by  $c'\ \varphi 1$  using  $\varphi\ c'\ \text{no-equiv no-imp simpleN } IH\varphi 1\ \text{c-in-c'-only}$  by auto
  moreover have super-grouped-by  $c'\ \varphi 2$ 
  using  $\varphi\ c'\ \text{no-equiv no-imp simpleN } IH\varphi 2\ \text{c-in-c'-only}$  by auto
  ultimately have  $?S\ \varphi$ 
  using super-grouped-by.intros(2)  $\varphi$  by (metis c wf-conn-helper-facts(5,6))
}
moreover {
  assume  $\varphi: \varphi = \text{conn}\ c\ [\varphi 1, \varphi 2] \wedge \text{wf-conn}\ c\ [\varphi 1, \varphi 2]$ 
  then have only-c-inside  $c\ \varphi 1 \wedge \text{only-c-inside}\ c\ \varphi 2$ 
  using c-in-c'-symb-c-implies-only-c-inside  $c'\ c'\ \text{c-in-c'-only list.set-intros}(1)$ 
  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')
  then have only-c-inside  $c\ (\text{conn}\ c\ [\varphi 1, \varphi 2])$ 
  unfolding only-c-inside-def using  $\varphi$ 
  by (simp add: only-c-inside-into-only-c-inside all-subformula-st-decomp)
  then have grouped-by  $c\ \varphi$  using  $\varphi\ \text{only-c-inside-imp-grouped-by}\ c$  by blast
  then have  $?S\ \varphi$  using super-grouped-by.intros(1) by metis
}
ultimately show  $?S\ \varphi$  by (metis  $\varphi'\ c\ c'\ cc'\ \text{conn.simps}$ (5,6) wf-conn-helper-facts(5,6))
qed

```

9.2 Conjunctive Normal Form

definition *is-conj-with-TF* **where** $\text{is-conj-with-TF} == \text{super-grouped-by } COr\ CAnd$

lemma *or-in-and-only-conjunction-in-disj*:

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

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

9.2.1 Full CNF transformation

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

definition *cnf-rew* **where** $\text{cnf-rew} =$
 (*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*

then have *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*

then have *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*

then have *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*

then have *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*

then have *no-Disj: or-in-and-only \varphiDisj*

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

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

simple-not \varphiDisj

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

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

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

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

qed

9.3 Disjunctive Normal Form

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

lemma *and-in-or-only-conjunction-in-disj:*

shows *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 propo \Rightarrow bool where*

is-dnf $\varphi \iff$ 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 $\varphi \varphi' \implies \text{is-dnf } \varphi'$

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

by (*meson and-in-or-only-conjunction-in-disj elimTB-full-propo-rew-step elimTB-inv(1,2) elim-imp-inv is-dnf-def no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(1-4) pushNeg-full-propo-rew-step pushNeg-inv(1-3)*)

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

10.1 Transformation

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

inductive *elimTBFull* **where**

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

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

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

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

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

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

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

The transformation is still consistent.

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

proof –

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

then show *?thesis* **using** *preserves-un-sat-def* **by** *auto*
qed

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

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

case (*nullary* φ')

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

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

next

case (*unary* ψ)

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

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

next

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

then have $\psi1 = FT \vee \psi2 = FT \vee \psi1 = FF \vee \psi2 = FF$

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

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

qed

The same applies here. We do not need the assumption, but the deep link between $\neg \text{no-T-F-except-top-level}$ 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*: $\neg \text{no-T-F-except-top-level } \varphi$

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

proof –

have *test-symb-false-nullary*:

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

by *auto*

moreover {

fix $c :: 'v \text{ connective}$ **and** $l :: 'v \text{ propo list}$ **and** $\psi :: 'v \text{ propo}$

have $H: \text{elimTBFull } (\text{conn } c \ l) \ \psi \implies \neg \text{no-T-F-symb-except-toplevel } (\text{conn } c \ l)$

by (*cases* (*conn* $c \ l$) *rule: elimTBFull.cases*) *auto*

}

ultimately show *?thesis*

using *no-test-symb-step-exists*[*of no-T-F-symb-except-toplevel* φ *elimTBFull*] *noTB*

no-T-F-symb-except-toplevel-step-exists' **unfolding** *no-T-F-except-top-level-def* **by** *metis*

qed

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

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

assumes *full* (*propo-rew-step elimTBFull*) $\varphi \psi$

shows *no-T-F-except-top-level* ψ

using *full-propo-rew-step-subformula no-T-F-except-top-level-rew' assms* **by** *fastforce*

10.2 More invariants

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

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

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

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

assume *a1*: *no-T-F* φ'

assume *a2*: *elim-equiv* $\varphi' \psi'$

have $\forall x0 \ x1. (\neg \text{elim-equiv } (x1 :: 'v \text{ propo}) \ x0 \vee (\exists v2 \ v3 \ v4 \ v5 \ v6 \ v7. x1 = \text{FEq } v2 \ v3 \wedge x0 = \text{FAnd } (\text{FImp } v4 \ v5) (\text{FImp } v6 \ v7) \wedge v2 = v4 \wedge v4 = v7 \wedge v3 = v5 \wedge v3 = v6))$
 $= (\neg \text{elim-equiv } x1 \ x0 \vee (\exists v2 \ v3 \ v4 \ v5 \ v6 \ v7. x1 = \text{FEq } v2 \ v3 \wedge x0 = \text{FAnd } (\text{FImp } v4 \ v5) (\text{FImp } v6 \ v7) \wedge v2 = v4 \wedge v4 = v7 \wedge v3 = v5 \wedge v3 = v6))$
by *meson*

then have $\forall p \ pa. \neg \text{elim-equiv } (p :: 'v \text{ propo}) \ pa \vee (\exists pb \ pc \ pd \ pe \ pf \ pg. p = \text{FEq } pb \ pc \wedge pa = \text{FAnd } (\text{FImp } pd \ pe) (\text{FImp } pf \ pg) \wedge pb = pd \wedge pd = pg \wedge pc = pe \wedge pc = pf)$

using *elim-equiv.cases* **by** *force*

then show *no-T-F* ψ' **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* $\varphi \implies \text{no-T-F } \varphi'$

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

and *no-T-F*: *no-T-F* (*conn* $c (\xi @ \varphi \# \xi')$)

{

assume *c*: $c = \text{CNot}$

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

then have *no-T-F* φ **using** *no-T-F c no-T-F-decomp-not* **by** *auto*

then have *no-T-F* (*conn* $c (\xi @ \varphi' \# \xi')$) **using** *c empty no-T-F-comp-not IH* **by** *auto*

}

moreover {

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

obtain *a b* **where** $\xi @ \varphi \# \xi' = [a, b]$

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

then have $\varphi: \varphi = a \vee \varphi = b$

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*

then have $\varphi': \text{no-T-F } \varphi'$ **using** *ab IH* φ **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)*)

then have $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \text{no-T-F } \zeta$ **using** $\zeta \varphi' ab$ **by** *fastforce*

moreover

```

have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \zeta \neq FT \wedge \zeta \neq FF$ 
  using  $\zeta \text{ corr no-T-F no-T-F-except-top-level-false no-T-F-no-T-F-except-top-level}$  by blast
then have  $\text{no-T-F-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
  by (metis  $\varphi' l'$  ab all-subformula-st-test-symb-true-phi  $c \text{ list.distinct}(1)$ 
     $\text{list.set-intros}(1,2)$  no-T-F-symb-except-toplevel-bin-decom
     $\text{no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false}(1,2)$  no-T-F-def wf-conn-binary
     $\text{wf-conn-list}(1,2))$ 
ultimately have  $\text{no-T-F } (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
  by (metis  $l'$  all-subformula-st-decomp-imp  $c \text{ no-T-F-def wf-conn-binary}$ )
}
moreover {
  fix  $x$ 
  assume  $c = CVar\ x \vee c = CF \vee c = CT$ 
  then have False using corr by auto
  then have  $\text{no-T-F } (\text{conn } c (\xi @ \varphi' \# \xi'))$  by auto
}
ultimately show  $\text{no-T-F } (\text{conn } c (\xi @ \varphi' \# \xi'))$  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  $\text{no-T-F-except-top-level } \varphi$ 
shows  $\text{no-T-F-except-top-level } \psi$ 
proof –
{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have  $\text{propo-rew-step elim-equiv } \varphi \psi \implies \text{no-T-F-except-top-level } \varphi$ 
     $\implies \text{no-T-F-except-top-level } \psi$ 
  proof –
    assume rel: propo-rew-step elim-equiv  $\varphi \psi$ 
    and no:  $\text{no-T-F-except-top-level } \varphi$ 
    {
      assume  $\varphi = FT \vee \varphi = FF$ 
      from rel this have False
      apply (induct rule: propo-rew-step.induct, auto simp: wf-conn-list(1,2))
      using elim-equiv.simps by blast+
      then have  $\text{no-T-F-except-top-level } \psi$  by blast
    }
    moreover {
      assume  $\varphi \neq FT \wedge \varphi \neq FF$ 
      then have  $\text{no-T-F } \varphi$ 
      by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
      then have  $\text{no-T-F } \psi$  using propo-rew-step-ElimEquiv-no-T-F rel by blast
      then have  $\text{no-T-F-except-top-level } \psi$  by (simp add: no-T-F-no-T-F-except-top-level)
    }
    ultimately show  $\text{no-T-F-except-top-level } \psi$  by metis
  qed
}
moreover {
  fix  $c :: 'v \text{ connective}$  and  $\xi \xi' :: 'v \text{ propo list}$  and  $\zeta \zeta' :: 'v \text{ propo}$ 
  assume rel: propo-rew-step elim-equiv  $\zeta \zeta'$ 
  and incl:  $\zeta \preceq \varphi$ 
  and corr:  $\text{wf-conn } c (\xi @ \zeta \# \xi')$ 
  and no-T-F:  $\text{no-T-F-symb-except-toplevel } (\text{conn } c (\xi @ \zeta \# \xi'))$ 
  and n:  $\text{no-T-F-symb-except-toplevel } \zeta'$ 

```

```

have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
proof
  have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
    using corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F
    by blast
  have l: ∀ φ ∈ set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
    using corr wf-conn-no-T-F-symb-iff p by blast
  from rel incl have ζ' ≠ FT ∧ ζ' ≠ FF
    apply (induction ζ ζ' rule: propo-rew-step.induct)
    apply (cases rule: elim-equiv.cases, auto simp: 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)+
  then have ∀ φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
  moreover have c ≠ CT ∧ c ≠ CF using corr by auto
  ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
    by (metis corr 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 ψ
  using full-propo-rew-step-inv-stay-with-inc[of elim-equiv no-T-F-symb-except-toplevel φ]
  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 φ ψ ⇒ no-T-F φ ⇒ no-T-F ψ
proof (induct rule: propo-rew-step.induct)
  case (global-rel φ' ψ')
  then show no-T-F ψ'
    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 φ φ' c ξ ξ')
  note rel = this(1) and IH = this(2) and corr = this(3) and no-T-F = this(4)
  {
    assume c: c = CNot
    then have empty: ξ = [] ξ' = [] using corr by auto
    then have no-T-F φ using no-T-F c no-T-F-decomp-not by auto
    then have no-T-F (conn c (ξ @ φ' # ξ')) using c empty no-T-F-comp-not IH by auto
  }
  moreover {
    assume c: c ∈ binary-connectives
    then obtain a b where ab: ξ @ φ # ξ' = [a, b]
      using corr list-length2-decomp wf-conn-bin-list-length by metis
    then have φ: φ = a ∨ φ = 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 ζ: ∀ ζ ∈ set (ξ @ φ # ξ'). no-T-F ζ using ab c propo-rew-one-step-lift.prem by auto

    then have φ': no-T-F φ'
      using ab IH φ corr no-T-F no-T-F-def all-subformula-st-decomp-explicit by auto
    have χ: ξ @ φ' # ξ' = [φ', b] ∨ ξ @ φ' # ξ' = [a, φ]
      by (metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps(2)
          butlast-append list.distinct(1) list.sel(3))
    then have ∀ ζ ∈ set (ξ @ φ' # ξ'). no-T-F ζ using ζ φ' ab by fastforce
    moreover

```



```

  have no-T-F (last (ξ @ φ' # ξ')) by (simp add: calculation)
  then have no-T-F-symb (conn c (ξ @ φ' # ξ'))
    by (metis χ φ' ζ 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 (ξ @ φ' # ξ')) using c χ by fastforce
}
moreover {
  fix x
  assume c = CVar x ∨ c = CF ∨ c = CT
  then have False using corr by auto
  then have no-T-F (conn c (ξ @ φ' # ξ')) by auto
}
ultimately show no-T-F (conn c (ξ @ φ' # ξ')) using corr wf-conn.cases by blast
qed

```

lemma *elim-imp-inv'*:

```

  fixes φ ψ :: 'v propo
  assumes full (propo-rew-step elim-imp) φ ψ and no-T-F-except-top-level φ
  shows no-T-F-except-top-level ψ
proof -
  {
    {
      fix φ ψ :: 'v propo
      have H: elim-imp φ ψ ⟹ no-T-F-except-top-level φ ⟹ no-T-F-except-top-level ψ
        by (induct φ ψ rule: elim-imp.induct, auto)
    } note H = this
    fix φ ψ :: 'v propo
    have propo-rew-step elim-imp φ ψ ⟹ no-T-F-except-top-level φ ⟹ no-T-F-except-top-level ψ
    proof -
      assume rel: propo-rew-step elim-imp φ ψ
      and no: no-T-F-except-top-level φ
      {
        assume φ = FT ∨ φ = FF
        from rel this have False
        apply (induct rule: propo-rew-step.induct)
        by (cases rule: elim-imp.cases, auto simp: wf-conn-list(1,2))
        then have no-T-F-except-top-level ψ by blast
      }
      moreover {
        assume φ ≠ FT ∧ φ ≠ FF
        then have no-T-F φ
          by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
        then have no-T-F ψ
          using rel propo-rew-step-ElimImp-no-T-F by blast
        then have no-T-F-except-top-level ψ by (simp add: no-T-F-no-T-F-except-top-level)
      }
      ultimately show no-T-F-except-top-level ψ by metis
    qed
  }
}
moreover {
  fix c :: 'v connective and ξ ξ' :: 'v propo list and ζ ζ' :: 'v propo
  assume rel: propo-rew-step elim-imp ζ ζ'
  and incl: ζ ≤ φ
  and corr: wf-conn c (ξ @ ζ # ξ')

```

```

and no-T-F: no-T-F-symb-except-toplevel (conn c (ξ @ ζ # ξ'))
and n: no-T-F-symb-except-toplevel ζ'
have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
proof
  have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
    by (simp add: corr no-T-F no-T-F-symb-except-toplevel-no-T-F-symb wf-conn-list(1,2))

  have l: ∀ φ ∈ set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
    using corr wf-conn-no-T-F-symb-iff p by blast
  from rel incl have ζ' ≠ FT ∧ ζ' ≠ FF
    apply (induction ζ ζ' rule: propo-rew-step.induct)
    apply (cases rule: 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))+
  then have ∀ φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
  moreover have c ≠ CT ∧ c ≠ CF using corr by auto
  ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
    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 ψ
  using full-propo-rew-step-inv-stay-with-inc[of elim-imp no-T-F-symb-except-toplevel φ]
  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-rew'* :: 'a propo ⇒ 'a propo ⇒ bool **where**

```

dnf-rew' =
  (full (propo-rew-step elimTBFULL)) OO
  (full (propo-rew-step elim-equiv)) OO
  (full (propo-rew-step elim-imp)) OO
  (full (propo-rew-step pushNeg)) OO
  (full (propo-rew-step pushConj))

```

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

```

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 φ' ⇒ 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 ⇒ 'a propo ⇒ bool **where**

```

cnf-rew' =
  (full (propo-rew-step elimTBFULL)) OO
  (full (propo-rew-step elim-equiv)) OO
  (full (propo-rew-step elim-imp)) OO

```

(full (propo-rew-step pushNeg)) OO
 (full (propo-rew-step pushDisj))

lemma *cnf-rew'-consistent: preserves-un-sat cnf-rew'*
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*
 $I \models_l L \longleftrightarrow L \in I$

declare *true-lit-def[simp]*

11.2.1 Consistency

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

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

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

lemma *consistent-interp-subset:*
assumes
 $A \subseteq B$ and
consistent-interp B
shows *consistent-interp A*
using *assms unfolding consistent-interp-def by auto*

lemma *consistent-interp-change-insert*:

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

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

$a \notin A \implies \text{consistent-interp } (\text{insert } a A) \longleftrightarrow \text{consistent-interp } A \wedge -a \notin A$
unfolding *consistent-interp-def* **by** *auto*

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

$\text{consistent-interp } A \implies a \notin A \implies -a \notin A \implies \text{consistent-interp } (\text{insert } a A)$
unfolding *consistent-interp-def* **by** *auto*

11.2.2 Atoms

definition *atms-of-ms* :: 'a literal multiset set \Rightarrow 'a set **where**

atms-of-ms $\psi s = \bigcup (\text{atms-of } ' \psi s)$

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

$\text{atms-of } (\text{mset } a) = \text{atm-of } ' \text{ set } a$
by (*induct a*) *auto*

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

$\text{atms-of-ms } (\text{mset } ' b) = (\bigcup x \in b. \text{atm-of } ' \text{ set } x)$
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]*:

$\text{atms-of-ms } \{\} = \{\}$
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-mempty[simp]*:

$\text{atms-of-ms } \{\{\#\}\} = \{\}$
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-mono*:

$A \subseteq B \implies \text{atms-of-ms } A \subseteq \text{atms-of-ms } B$
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-finite[simp]*:

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

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

$\text{atms-of-ms } (\psi s \cup \chi s) = \text{atms-of-ms } \psi s \cup \text{atms-of-ms } \chi s$
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-insert[simp]*:

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

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

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

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

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

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

lemma *atms-of-ms-empty-iff*:
 $\text{atms-of-ms } A = \{\} \longleftrightarrow A = \{\{\#\}\} \vee A = \{\}$
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 $\text{atm-of } L \in \text{atms-of-ms } N$
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 $\text{atm-of } L \in \text{atms-of-ms } N$
using *in-implies-atm-of-on-atms-of-ms[of - C + \{\#L\# \}] assms* **by** *auto*

lemma *in-m-in-literals*:
assumes $\{\#A\# \} + D \in \psi$
shows $\text{atm-of } A \in \text{atms-of-ms } \psi$
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]*:
 $\text{atms-of-s } (\text{insert } L \ Ib) = \{\text{atm-of } L\} \cup \text{atms-of-s } Ib$
unfolding *atms-of-s-def* **by** *auto*

lemma *in-atms-of-s-decomp*[iff]:
 $P \in \text{atms-of-s } I \iff (Pos\ P \in I \vee Neg\ P \in I)$ (is ?P \iff ?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 interp \Rightarrow 'a set \Rightarrow bool **where**
total-over-set $I\ S = (\forall l \in S. Pos\ l \in I \vee Neg\ l \in I)$

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

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

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

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

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

lemma *total-over-set-union*[iff]:
 $\text{total-over-set } I\ (Ls \cup Ls') \iff (\text{total-over-set } I\ Ls \wedge \text{total-over-set } I\ Ls')$
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-m-subset*:
 $A \subseteq B \implies \text{total-over-m } I\ B \implies \text{total-over-m } I\ A$
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 $\text{total-over-m } I\ \{C + D\} \iff (\text{total-over-m } I\ \{C\} \wedge \text{total-over-m } I\ \{D\})$
using *assms* **unfolding** *total-over-m-def* *total-over-set-def* **by** *auto*

lemma *total-over-m-union*[iff]:
 $\text{total-over-m } I\ (A \cup B) \iff (\text{total-over-m } I\ A \wedge \text{total-over-m } I\ B)$
unfolding *total-over-m-def* *total-over-set-def* **by** *auto*

lemma *total-over-m-insert*[iff]:
 $\text{total-over-m } I\ (\text{insert } a\ A) \iff (\text{total-over-set } I\ (\text{atms-of } a) \wedge \text{total-over-m } I\ A)$

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

lemma *total-over-m-extension:*

fixes $I :: 'v$ literal set **and** $A :: 'v$ clauses

assumes *total: total-over-m I A*

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

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

proof $-$

let $?I' = \{Pos\ v \mid v. v \in \text{atms-of-ms } B \wedge v \notin \text{atms-of-ms } A\}$

have $(\forall x \in ?I'. \text{atm-of } x \in \text{atms-of-ms } B \wedge \text{atm-of } x \notin \text{atms-of-ms } A)$ **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 $\exists I'. \text{total-over-m } (I \cup I') (A \cup B)$

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

proof $-$

let $?I' = \{Pos\ v \mid v. v \in \text{atms-of-ms } B \wedge v \notin \text{atms-of-ms } A \wedge Pos\ v \notin I \wedge Neg\ v \notin I\}$

have $(\forall x \in ?I'. \text{atm-of } x \in \text{atms-of-ms } B \wedge \text{atm-of } x \notin \text{atms-of-ms } A)$ **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) (rename-tac L, case-tac L, auto)*

ultimately show *?thesis* **by** *blast*

qed

lemma *total-over-set-atms-of-m[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 $A \in I \vee -A \in I$

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}) { ψ }*

and $L: \neg L \in \# \psi \neg L \notin \# \psi$

shows *total-over-m I { ψ }*

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

proof

fix l

assume $l: l \in \text{atms-of-ms } \{\psi\}$

then have $Pos\ l \in I \vee Neg\ l \in I \vee l = \text{atm-of } L$

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** (*cases L*) *auto*
 then show *False* **using** *L* **by** *auto*
 qed
 ultimately show $\text{Pos } l \in I \vee \text{Neg } l \in I$ **using** *l* **by** *metis*
 qed

lemma *total-union*:

assumes *total-over-m I ψ*
 shows *total-over-m (I ∪ I') ψ*
 using *assms unfolding total-over-m-def total-over-set-def* **by** *auto*

lemma *total-union-2*:

assumes *total-over-m I ψ*
 and *total-over-m I' ψ'*
 shows *total-over-m (I ∪ I') (ψ ∪ ψ')*
 using *assms unfolding total-over-m-def total-over-set-def* **by** *auto*

11.2.4 Interpretations

definition *true-cls* :: '*a interp* ⇒ '*a clause* ⇒ *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-if-split-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*: $\text{set-mset } C \subseteq \text{set-mset } D \implies I \models C \implies I \models D$
 unfolding *true-cls-def subset-eq Bex-def* **by** *metis*

lemma *true-cls-mono-leD[dest]*: $A \subseteq \# B \implies I \models A \implies I \models B$
 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-cls-true-cls*:
 assumes $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$

and $\text{atms-of } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models L$
shows $I \models L$
using *assms* **unfolding** *true-cls-def true-lit-def Bex-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-cls-true-cls subset-trans*)

11.2.5 Satisfiability

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

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

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

have $I\text{-}CC: ?I \models_s CC$
using $I\text{-}CC$ *in-implies-atm-of-on-atms-of-ms* **unfolding** *true-clss-def Ball-def true-cls-def Bex-def true-lit-def*
by *blast*

moreover have $\text{cons: consistent-interp } ?I$
using cons **unfolding** *consistent-interp-def* **by** *auto*
moreover have $\text{total-over-m } ?I \text{ } CC$
using tot **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
moreover

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

11.2.6 Entailment for Multisets of Clauses

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

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

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

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

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

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

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

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

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

theorem *true-clss-remove-unused*:
assumes $I \models_s \psi$
shows $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \models_s \psi$
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-cls-remove-unused[of I]*)
moreover have $\{v \in I. \text{atm-of } v \in \text{atms-of } x\} \subseteq \{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\}$
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-cls-mono-set-mset-l* **by** *blast*

qed

A simple application of the previous theorem:

lemma *true-clss-union-decrease*:

assumes $II': I \cup I' \models \psi$

and $H: \forall v \in I'. \text{atm-of } v \notin \text{atms-of } \psi$

shows $I \models \psi$

proof –

let $?I = \{v \in I \cup I'. \text{atm-of } v \in \text{atms-of } \psi\}$

have $?I \models \psi$ **using** *true-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 $\text{consI}: \text{consistent-interp } I$

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

and $\text{consA}: \text{consistent-interp } A$

shows $\text{consistent-interp } (A \cup I)$

proof (*rule ccontr*)

assume $\neg ?thesis$

moreover have $\bigwedge L. \neg (L \in A \wedge \neg L \in I)$

using *disj unfolding atms-of-s-def* **by** (*auto simp add: rev-image-eqI*)

ultimately show *False*

using *consA consI unfolding consistent-interp-def* **by** (*metis (full-types) Un-iff literal.exhaust-sel uminus-Neg uminus-Pos*)

qed

lemma *total-remove-unused*:

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

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

using *assms unfolding total-over-m-def total-over-set-def*

by (*metis (lifting) literal.sel(1,2) mem-Collect-eq*)

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

assumes $\text{insert } a \ M' \models D$

and $\text{atm-of } a \notin \text{atms-of } D$

shows $M' \models D$

using *assms* **by** (*auto simp add: atm-of-lit-in-atms-of true-clss-def*)

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

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

shows $\text{total-over-set } I \ K \longleftrightarrow (\forall l \in K. l \in (\text{atm-of } I))$

unfolding *total-over-set-def* **by** (*metis atms-of-s-def in-atms-of-s-decomp*)

11.2.7 Tautologies

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

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

assumes $\text{Pos } p \in \# A$ **and** $\text{Neg } p \in \# A$

shows *tautology* A

using *assms unfolding tautology-def total-over-set-def true-cls-def Bex-def*

by (*metis atms-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* $?I$ (*atms-of* ψ)

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-def* **apply** *clarify*

using p **by** (*rename-tac x L, 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 $(\{\#a\# \} + L) \longleftrightarrow \text{tautology } L \vee -a \in \# L$

unfolding *tautology-decomp* **by** (*cases a*) *auto*

lemma *minus-interp-tautology*:

assumes $\{-L \mid L. L \in \# \chi\} \models \chi$

shows *tautology* 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 $I \cup \{\text{Pos } P\} \models \varphi$

and $I \cup \{\text{Neg } P\} \models \varphi$

shows $I \models \varphi \vee \text{tautology } \varphi$

using *assms unfolding true-cls-def* **by** *auto*

lemma *tautology-imp-tautology*:
fixes $\chi \ \chi' :: 'v \text{ clause}$
assumes $\forall I. \text{total-over-m } I \ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$ **and** *tautology* χ
shows *tautology* χ' **unfolding** *tautology-def*
proof (*intro allI HOL.impI*)
fix $I :: 'v \text{ literal set}$
assume *totI*: *total-over-set* $I \ (\text{atms-of } \chi')$
let $?I' = \{ \text{Pos } v \mid v. v \in \text{atms-of } \chi \wedge v \notin \text{atms-of-s } I \}$
have *totI'*: *total-over-m* $(I \cup ?I') \ \{\chi\}$ **unfolding** *total-over-m-def* *total-over-set-def* **by** *auto*
then have $\chi: I \cup ?I' \models \chi$ **using** *assms(2)* **unfolding** *total-over-m-def* *tautology-def* **by** *simp*
then have $I \cup (?I' - I) \models \chi'$ **using** *assms(1)* *totI'* **by** *auto*
moreover have $\bigwedge L. L \in \# \chi' \implies L \notin ?I'$
using *totI* **unfolding** *total-over-set-def* **by** (*auto dest: pos-lit-in-atms-of*)
ultimately show $I \models \chi'$ **unfolding** *true-cls-def* **by** *auto*
qed

11.2.8 Entailment for clauses and propositions

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

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

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

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

lemma *true-cls-cls-refl[simp]*:
 $A \models_f A$
unfolding *true-cls-cls-def* **by** *auto*

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

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

lemma *true-prop-true-clause[iff]*:
 $\{\varphi\} \models_p \psi \iff \varphi \models_f \psi$
unfolding *true-cls-cls-def* *true-clss-cls-def* **by** *auto*

lemma *true-clss-clss-true-clss-cls[iff]*:
 $N \models_{ps} \{\psi\} \iff N \models_p \psi$
unfolding *true-clss-clss-def* *true-clss-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-clss-subset*:
 $A \subseteq B \implies A \models_p CC \implies B \models_p CC$
unfolding *true-clss-clss-def total-over-m-union* **by** (*simp add: total-over-m-subset true-clss-mono*)

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

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

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

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

lemma *true-clss-clss-union-l[simp]*:
 $A \models_{ps} CC \implies A \cup B \models_{ps} CC$
unfolding *true-clss-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-clss-union-l-r[simp]*:
 $B \models_{ps} CC \implies A \cup B \models_{ps} CC$
unfolding *true-clss-clss-def total-over-m-union* **by** *fastforce*

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

lemma *true-clss-clss-insert-l[simp]*:
 $A \models_p C \implies \text{insert } a \ A \models_p C$
unfolding *true-clss-clss-def true-clss-def* **using** *total-over-m-union*
by (*metis Un-iff insert-is-Un sup commute*)

lemma *true-clss-clss-insert-l[simp]*:
 $A \models_{ps} C \implies \text{insert } a \ A \models_{ps} C$
unfolding *true-clss-clss-def true-clss-clss-def true-clss-def* **by** *blast*

lemma *true-clss-clss-union-and[iff]*:
 $A \models_{ps} C \cup D \iff (A \models_{ps} C \wedge A \models_{ps} D)$

proof
 $\{$
fix $A \ C \ D :: 'a \ \text{clauses}$
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 $\text{totAC}: \text{total-over-m } I \ (A \cup C)$

```

    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  $\models_{ps}$  insert L Ls  $\longleftrightarrow$  (A  $\models_p$  L  $\wedge$  A  $\models_{ps}$  Ls)
  using true-clss-clss-union-and[of A {L} Ls] by auto

lemma true-clss-clss-subset:
  A  $\subseteq$  B  $\implies$  A  $\models_{ps}$  CC  $\implies$  B  $\models_{ps}$  CC
  by (metis subset-Un-eq true-clss-clss-union-l)

lemma union-trus-clss-clss[simp]: A  $\cup$  B  $\models_{ps}$  B
  unfolding true-clss-clss-def by auto

lemma true-clss-clss-remove[simp]:
  A  $\models_{ps}$  B  $\implies$  A  $\models_{ps}$  B - C
  by (metis Un-Diff-Int true-clss-clss-union-and)

lemma true-clss-clss-subsetE:
  N  $\models_{ps}$  B  $\implies$  A  $\subseteq$  B  $\implies$  N  $\models_{ps}$  A
  by (metis sup.orderE true-clss-clss-union-and)

lemma true-clss-clss-in-imp-true-clss-clss:
  assumes N  $\models_{ps}$  U
  and A  $\in$  U
  shows N  $\models_p$  A
  using assms mk-disjoint-insert by fastforce

lemma all-in-true-clss-clss:  $\forall x \in B. x \in A \implies A \models_{ps} B$ 
  unfolding true-clss-clss-def true-clss-def by auto

lemma true-clss-clss-left-right:
  assumes A  $\models_{ps}$  B
  and A  $\cup$  B  $\models_{ps}$  M
  shows A  $\models_{ps}$  M  $\cup$  B
  using assms unfolding true-clss-clss-def by auto

```


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

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

proof –

assume $a1: A \cup C \models_{ps} D$

assume $B \models_{ps} C$

then have $f2: \bigwedge M. M \cup B \models_{ps} C$

by (*meson true-clss-clss-union-l-r*)

have $\bigwedge M. C \cup (M \cup A) \models_{ps} D$

using $a1$ **by** (*simp add: Un-commute sup-left-commute*)

then show *?thesis*

using $f2$ **by** (*metis (no-types) Un-commute true-clss-clss-left-right true-clss-clss-union-and*)

qed

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

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

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

shows $N \models_p D + C$

unfolding *true-clss-clss-def*

proof (*intro allI impI*)

fix I

assume *tot*: *total-over-m* $I (N \cup \{D + C\})$

and *consistent-interp* I

and $I \models_s N$

{

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

then have *total-over-m* $I \{D + \{\#- L\# \}\}$

using *tot* **by** (*cases L*) *auto*

then have $I \models D + \{\#- L\# \}$ **using** $D \langle I \models_s N \rangle$ *tot* *consistent-interp I*

unfolding *true-clss-clss-def* **by** *auto*

moreover

have *total-over-m* $I \{C + \{\#L\# \}\}$

using L *tot* **by** (*cases L*) *auto*

then have $I \models C + \{\#L\# \}$

using $C \langle I \models_s N \rangle$ *tot* *consistent-interp I* **unfolding** *true-clss-clss-def* **by** *auto*

ultimately have $I \models D + C$ **using** *consistent-interp I* *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** $L \langle$ *consistent-interp I* \rangle **by** *auto*

moreover have *total-over-m* $?I' \{D + \{\#- L\# \}\}$

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

moreover have *total-over-m* $?I' N$ **using** *tot* **using** *total-union* **by** *blast*

moreover have $?I' \models_s N$ **using** $I \models_s N$ **using** *true-clss-union-increase* **by** *blast*

ultimately have $?I' \models D + \{\#- L\# \}$

using D **unfolding** *true-clss-clss-def* **by** *blast*

then have $?I' \models D$ **using** L **by** *auto*

moreover

have *total-over-set* $I (atms-of (D + C))$ **using** *tot* **by** *auto*

then have $L \notin \# D \wedge -L \notin \# D$

using L **unfolding** *total-over-set-def atms-of-def* **by** (*cases L*) *force+*

ultimately have $I \models D + C$ **unfolding** *true-clss-clss-def* **by** *auto*

}

ultimately show $I \models D + C$ **by** *blast*

qed

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*

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 *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 *consistent-interp* $I \rangle$ **unfolding** *true-clss-cls-def* **by** *auto*

ultimately have $I \models D \# \cup C$ **using** \langle *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** $L \langle$ *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*]:

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

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) (rename-tac L, 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-clss-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 *atms-of-replicate-mset-replicate-mset-uminus[simp]:*

```

 $\text{atms-of } (D - \text{replicate-mset } (\text{count } D\ L)\ L - \text{replicate-mset } (\text{count } D\ (-L))\ (-L))$ 
   $= \text{atms-of } D - \{\text{atm-of } L\}$ 
by (fastforce simp: 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-clss-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 total-over-m  $I\ \{?D'\}$ 
  then have tot: 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 ⊨ ?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': total-over-m (I ∪ {-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 remove-literal-in-model-tautology by force
} note  $H' = \text{this}$ 

have  $L \notin\# C$  and  $-L \notin\# C$  using L 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: subseteq-mset-def not-in-iff)
have  $\text{card } \{ \text{Pos } v \mid v. v \in \text{atms-of } ?D' \wedge v \notin \text{atms-of } C \} <$ 
   $\text{card } \{ \text{Pos } v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C \}$ 
  using L by (auto intro!: psubset-card-mono)
then show ?case
  using IH C-in-D' H' unfolding card[symmetric] by blast
qed

```

11.4 Removing Duplicates

```

lemma tautology-remdups-mset[iff]:
  tautology (remdups-mset C) ⟷ tautology C
  unfolding tautology-decomp by auto

lemma atms-of-remdups-mset[simp]: atms-of (remdups-mset C) = atms-of C
  unfolding atms-of-def by auto

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

```

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

lemma simple-clss-empty[simp]:
  simple-clss {} = {{\#}}
  unfolding simple-clss-def by auto

lemma simple-clss-insert:
  assumes  $l \notin \text{atms}$ 
  shows simple-clss (insert l atms) =
     $(\text{op } + \{ \# \text{Pos } l \# \}) \text{ ` } (\text{simple-clss atms})$ 
     $\cup (\text{op } + \{ \# \text{Neg } l \# \}) \text{ ` } (\text{simple-clss atms})$ 
     $\cup \text{simple-clss atms}(\text{is } ?I = ?U)$ 
proof (standard; standard)
  fix C

```

```

assume  $C \in ?I$ 
then have
   $atms: atms-of\ C \subseteq insert\ l\ atms$  and
   $taut: \neg tautology\ C$  and
   $dist: distinct-mset\ C$ 
  unfolding simple-clss-def by auto
have  $H: \bigwedge x. x \in \# C \implies atm-of\ x \in insert\ l\ atms$ 
  using atm-of-lit-in-atms-of atms by blast
consider
  (Add)  $L \text{ where } L \in \# C \text{ and } L = Neg\ l \vee L = Pos\ l$ 
  | (No)  $Pos\ l \notin \# C\ Neg\ l \notin \# C$ 
  by auto
then show  $C \in ?U$ 
proof cases
  case Add
    then have  $L \notin \# C - \{\#L\# \}$ 
    using dist unfolding distinct-mset-def by (auto simp: not-in-iff)
    moreover have  $-L \notin \# C$ 
    using taut Add by auto
    ultimately have  $atms-of\ (C - \{\#L\# \}) \subseteq atms$ 
    using atms Add by (smt H atms-of-def imageE in-diffD insertE literal.exhaust-sel
      subset-iff uminus-Neg uminus-Pos)

    moreover have  $\neg tautology\ (C - \{\#L\# \})$ 
    using taut by (metis Add(1) insert-DiffM tautology-add-single)
    moreover have  $distinct-mset\ (C - \{\#L\# \})$ 
    using dist by auto
    ultimately have  $(C - \{\#L\# \}) \in simple-clss\ atms$ 
    using Add unfolding simple-clss-def by auto
    moreover have  $C = \{\#L\# \} + (C - \{\#L\# \})$ 
    using Add by (auto simp: multiset-eq-iff)
    ultimately show ?thesis using Add by auto
  next
    case No
    then have  $C \in simple-clss\ atms$ 
    using taut atms dist unfolding simple-clss-def
    by (auto simp: atm-iff-pos-or-neg-lit split: if-split-asm dest!: H)
    then show ?thesis by blast
  qed
next
fix  $C$ 
assume  $C \in ?U$ 
then consider
  (Add)  $L\ C' \text{ where } C = \{\#L\# \} + C' \text{ and } C' \in simple-clss\ atms \text{ and }$ 
   $L = Pos\ l \vee L = Neg\ l$ 
  | (No)  $C \in simple-clss\ atms$ 
  by auto
then show  $C \in ?I$ 
proof cases
  case No
    then show ?thesis unfolding simple-clss-def by auto
  next
    case (Add L C') note  $C' = this(1)$  and  $C = this(2)$  and  $L = this(3)$ 
    then have
       $atms: atms-of\ C' \subseteq atms$  and

```

```

    taut:  $\neg$ tautology  $C'$  and
    dist: distinct-mset  $C'$ 
    unfolding simple-clss-def by auto
  have atms-of  $C \subseteq \text{insert } l \text{ atms}$ 
    using atms  $C' L$  by auto
  moreover have  $\neg$  tautology  $C$ 
    using taut  $C' L$  by (metis assms atm-of-lit-in-atms-of atms literal.sel(1,2) subset-eq
      tautology-add-single uminus-Neg uminus-Pos)
  moreover have distinct-mset  $C$ 
    using dist  $C' L$ 
    by (metis assms atm-of-lit-in-atms-of atms contra-subsetD distinct-mset-add-single
      literal.sel(1,2))
  ultimately show ?thesis unfolding simple-clss-def by blast
qed
qed

lemma simple-clss-finite:
  fixes atms :: 'v set
  assumes finite atms
  shows finite (simple-clss atms)
  using assms by (induction rule: finite-induct) (auto simp: simple-clss-insert)

lemma simple-clssE:
  assumes
     $x \in \text{simple-clss atms}$ 
  shows atms-of  $x \subseteq \text{atms} \wedge \neg$ tautology  $x \wedge$  distinct-mset  $x$ 
  using assms unfolding simple-clss-def by auto

lemma cls-in-simple-clss:
  shows  $\{\#\} \in \text{simple-clss } s$ 
  unfolding simple-clss-def by auto

lemma simple-clss-card:
  fixes atms :: 'v set
  assumes finite atms
  shows card (simple-clss atms)  $\leq (3::\text{nat}) \wedge$  (card atms)
  using assms
proof (induct atms rule: finite-induct)
  case empty
  then show ?case by auto
next
  case (insert  $l C$ ) note fin = this(1) and  $l = \text{this}(2)$  and IH = this(3)
  have notin:
     $\wedge C'. \{\#\text{Pos } l\# \} + C' \notin \text{simple-clss } C$ 
     $\wedge C'. \{\#\text{Neg } l\# \} + C' \notin \text{simple-clss } C$ 
    using  $l$  unfolding simple-clss-def by auto
  have H:  $\wedge C' D. \{\#\text{Pos } l\# \} + C' = \{\#\text{Neg } l\# \} + D \implies D \in \text{simple-clss } C \implies \text{False}$ 
  proof -
    fix  $C' D$ 
    assume  $C'D: \{\#\text{Pos } l\# \} + C' = \{\#\text{Neg } l\# \} + D$  and  $D: D \in \text{simple-clss } C$ 
    then have  $\text{Pos } l \in \# D$  by (metis insert-noteq-member literal.distinct(1) union-commute)
    then have  $l \in \text{atms-of } D$ 
      by (simp add: atm-iff-pos-or-neg-lit)
    then show False using  $D l$  unfolding simple-clss-def by auto
  qed
qed

```

```

let ?P = (op + {#Pos l#}) ‘ (simple-clss C)
let ?N = (op + {#Neg l#}) ‘ (simple-clss C)
let ?O = simple-clss C
have card (?P ∪ ?N ∪ ?O) = card (?P ∪ ?N) + card ?O
  apply (subst card-Un-disjoint)
  using l fin by (auto simp: simple-clss-finite notin)
moreover have card (?P ∪ ?N) = card ?P + card ?N
  apply (subst card-Un-disjoint)
  using l fin H by (auto simp: simple-clss-finite notin)
moreover
  have card ?P = card ?O
    using inj-on-iff-eq-card[of ?O op + {#Pos l#}]
    by (auto simp: fin simple-clss-finite inj-on-def)
  moreover have card ?N = card ?O
    using inj-on-iff-eq-card[of ?O op + {#Neg l#}]
    by (auto simp: fin simple-clss-finite inj-on-def)
  moreover have (3::nat) ^ card (insert l C) = 3 ^ (card C) + 3 ^ (card C) + 3 ^ (card C)
    using l by (simp add: fin mult-2-right numeral-3-eq-3)
  ultimately show ?case using IH l by (auto simp: simple-clss-insert)
qed

```

```

lemma simple-clss-mono:
  assumes incl: atms ⊆ atms'
  shows simple-clss atms ⊆ simple-clss atms'
  using assms unfolding simple-clss-def by auto

```

```

lemma distinct-mset-not-tautology-implies-in-simple-clss:
  assumes distinct-mset χ and ¬tautology χ
  shows χ ∈ simple-clss (atms-of χ)
  using assms unfolding simple-clss-def by auto

```

```

lemma simplified-in-simple-clss:
  assumes distinct-mset-set ψ and ∀ χ ∈ ψ. ¬tautology χ
  shows ψ ⊆ simple-clss (atms-of-ms ψ)
  using assms unfolding simple-clss-def
  by (auto simp: distinct-mset-set-def atms-of-ms-def)

```

11.6 Experiment: Expressing the Entailments as Locales

```

locale entail =
  fixes entail :: 'a set ⇒ 'b ⇒ bool (infix |=e 50)
  assumes entail-insert[simp]: I ≠ {} ⇒ insert L I |=e x ⟷ {L} |=e x ∨ I |=e x
  assumes entail-union[simp]: I |=e A ⇒ I ∪ I' |=e A
begin

```

```

definition entails :: 'a set ⇒ 'b set ⇒ bool (infix |=es 50) where
  I |=es A ⟷ (∀ a ∈ A. I |=e a)

```

```

lemma entails-empty[simp]:
  I |=es {}
  unfolding entails-def by auto

```

```

lemma entails-single[iff]:
  I |=es {a} ⟷ I |=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 $?J = J \cup \{\text{Pos } (\text{atm-of } P) | P. P \in \# \ C \wedge \text{atm-of } P \notin \text{atm-of } 'J\}$
have $I \subseteq ?J$ **using** $\langle I \subseteq J \rangle$ **by** *auto*
moreover **have** *consistent-interp* $?J$


```

    using cons unfolding consistent-interp-def apply (intro allI)
    by (rename-tac L, case-tac L) (fastforce simp add: image-iff)+
moreover have total-over-m ?J N
    using tot unfolding total-over-m-def total-over-set-def atms-of-ms-def
    apply clarify
    apply (rename-tac l a, case-tac a ∈ N - {C})
    apply auto[]
    using atms-of-s-def atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
    by (fastforce simp: atms-of-def)
ultimately have ?J ⊨s N
    using assms unfolding true-clss-ext-def by blast
then have ?J ⊨s N - {C} by auto
have  $\{v \in ?J. \text{atm-of } v \in \text{atms-of-ms } (N - \{C\})\} \subseteq J$ 
    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 (?J ⊨s N - {C})] 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 ⊨sext A
  shows satisfiable A
  by (metis Un-empty-left assms satisfiable-carac subset-Un-eq sup.left-idem
    total-over-m-consistent-extension total-over-m-empty true-clss-ext-def)

```

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

```

  assumes  $\neg \text{consistent-interp } I$ 
  shows  $I \not\models_{\text{sext}} A$ 
  by (meson assms consistent-interp-subset true-clss-ext-def)
end

```

theory *Prop-Resolution*

imports *Partial-Clausal-Logic List-More Wellfounded-More*

begin

12 Resolution

12.1 Simplification Rules

inductive *simplify* :: *'v clauses* \Rightarrow *'v clauses* \Rightarrow *bool* **for** *N* :: *'v clause set* **where**

tautology-deletion:

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

condensation:

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

subsumption:

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

lemma *simplify-preserves-un-sat'*:

```

  fixes N N' :: 'v clauses
  assumes simplify N N'
  and total-over-m I N
  shows  $I \models_s N' \longrightarrow I \models_s N$ 
  using assms
proof (induct rule: simplify.induct)

```

```

case (tautology-deletion A P)
then have  $I \models A + \{\#Pos\ P\} + \{\#Neg\ P\}$ 
  by (metis total-over-m-def total-over-set-literal-defined true-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  $I \models N - \{B\} \implies I \models A$  using  $\langle A \in N \rangle$  by (simp add: true-clss-def)
moreover have  $I \models A \implies I \models B$  using  $\langle A < \# B \rangle$  by auto
ultimately show ?case by (metis insert-Diff-single true-clss-insert)
qed

```

```

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

```

```

lemma simplify-preserves-un-sat'':
  fixes  $N\ N' :: 'v\ clauses$ 
  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  $N\ N' :: 'v\ clauses$ 
  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  $\psi \longleftrightarrow$  finite  $\psi'$ 
  using assms by (induct rule: simplify.induct, auto simp add: remove-def)

```

```

lemma rtrancp-simplify-preserves-finite:
  assumes rtrancp simplify  $\psi\ \psi'$ 
  shows finite  $\psi \longleftrightarrow$  finite  $\psi'$ 
  using assms by (induct rule: rtrancp-induct) (auto simp add: simplify-preserves-finite)

```

```

lemma simplify-atms-of-ms:
  assumes simplify  $\psi\ \psi'$ 
  shows  $atms-of-ms\ \psi' \subseteq atms-of-ms\ \psi$ 
  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  $A + \{\#P\} + \{\#P\} \in \psi \implies \exists x \in \psi. \text{atm-of } P \in \text{atm-of } x$ 
  by (metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff)
ultimately show ?case by (auto simp add: atms-of-def)
next
case (subsumption A P)
then show ?case by auto
qed

lemma rtrancpl-simplify-atms-of-ms:
  assumes rtrancpl_simplify  $\psi \ \psi'$ 
  shows  $\text{atms-of-ms } \psi' \subseteq \text{atms-of-ms } \psi$ 
  using assms apply (induct rule: rtrancpl-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 $\text{uncon-res} :: 'v \text{ uncon-state} \Rightarrow 'v \text{ uncon-state} \Rightarrow \text{bool}$ **where**
resolution:
 $\{\#Pos \ p\} + C \in N \implies \{\#Neg \ p\} + D \in N \implies (\{\#Pos \ p\} + C, \{\#Neg \ p\} + D) \notin \text{already-used}$
 $\implies \text{uncon-res } (N) (N \cup \{C + D\})$ |
factoring: $\{\#L\} + \{\#L\} + C \in N \implies \text{uncon-res } N (N \cup \{C + \{\#L\}\})$

lemma *uncon-res-increasing:*
 assumes $\text{uncon-res } S \ S'$ and $\psi \in S$
 shows $\psi \in S'$
 using assms by (induct rule: uncon-res.induct) auto

lemma *rtrancpl-uncon-inference-increasing:*
 assumes $\text{rtrancpl uncon-res } S \ S'$ and $\psi \in S$
 shows $\psi \in S'$
 using assms by (induct rule: rtrancpl-induct) (auto simp add: uncon-res-increasing)

12.2.1 Subsumption

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

lemma *subsumes-refl[simp]:*
 $\text{subsumes } \chi \ \chi$

unfolding *subsumes-def* **by** *auto*

lemma *subsumes-subsumption*:

assumes *subsumes* $D \chi$

and $C \subset\# D$ **and** $\neg \text{tautology } \chi$

shows *subsumes* $C \chi$ **unfolding** *subsumes-def*

using *assms* *subsumption-total-over-m* *subsumption-chained* **unfolding** *subsumes-def*

by (*blast intro!*: *subset-mset.less-imp-le*)

lemma *subsumes-tautology*:

assumes *subsumes* $(C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}) \chi$

shows *tautology* 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:

$\{\#Pos\ p\#\} + C \in N \implies \{\#Neg\ p\#\} + D \in N \implies (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$
already-used

$\implies \text{inference-clause } (N, \text{already-used}) (C + D, \text{already-used} \cup \{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\}) \mid$

factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \implies \text{inference-clause } (N, \text{already-used}) (C + \{\#L\#\}, \text{already-used})$

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

inference-step: *inference-clause* S (*clause*, *already-used*)

$\implies \text{inference } S (\text{fst } S \cup \{\text{clause}\}, \text{already-used})$

abbreviation *already-used-inv*

:: *'a literal multiset set* \times (*'a literal multiset* \times *'a literal multiset*) *set* \Rightarrow *bool* **where**

already-used-inv state \equiv

$(\forall (A, B) \in \text{snd state}. \exists p. \text{Pos } p \in\# A \wedge \text{Neg } p \in\# B \wedge$

$((\exists \chi \in \text{fst state}. \text{subsumes } \chi ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$

$\vee \text{tautology } ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

lemma *inference-clause-preserves-already-used-inv*:

assumes *inference-clause* $S S'$

and *already-used-inv* S

shows *already-used-inv* ($\text{fst } S \cup \{\text{fst } S'\}$, *snd* S')

using *assms* **apply** (*induct rule*: *inference-clause.induct*)

by *fastforce*+

lemma *inference-preserves-already-used-inv*:

assumes *inference* $S S'$

and *already-used-inv* S

shows *already-used-inv* S'

using *assms*

proof (*induct rule*: *inference.induct*)

case (*inference-step* S *clause* *already-used*)

then show ?*case*

using *inference-clause-preserves-already-used-inv*[*of* S (*clause*, *already-used*)] **by** *simp*

qed

```

lemma rtrancp-inference-preserves-already-used-inv:
  assumes rtrancp inference S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms apply (induct rule: rtrancp-induct, simp)
  using inference-preserves-already-used-inv unfolding tautology-def by fast

lemma subsumes-condensation:
  assumes subsumes (C + {#L#} + {#L#}) D
  shows subsumes (C + {#L#}) D
  using assms unfolding subsumes-def by simp

lemma simplify-preserves-already-used-inv:
  assumes simplify N N'
  and already-used-inv (N, already-used)
  shows already-used-inv (N', already-used)
  using assms
proof (induct rule: simplify.induct)
  case (condensation C L)
  then show ?case
    using subsumes-condensation by simp fast
next
  {
    fix a:: 'a and A :: 'a set and P
    have  $(\exists x \in \text{Set.remove } a \ A. P \ x) \longleftrightarrow (\exists x \in A. x \neq a \wedge P \ x)$  by auto
  } note ex-member-remove = this
  {
    fix a a0 :: 'v clause and A :: 'v clauses and y
    assume  $a \in A$  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  $[\text{simp}]: x = (a, b)$ 
  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  $\chi \in N$   $\text{subsumes } \chi (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \}))$ 
     $\neg \text{tautology } (a - \{\#\text{Pos } p\# \} + (b - \{\#\text{Neg } p\# \}))$ 
  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 + {#Pos P#} + {#Neg P#} ∈ N
  assume already-used-inv (N, already-used)
  and (a, b) ∈ snd (N - {C + {#Pos P#} + {#Neg P#}}, already-used)
  then obtain p where
    Pos p ∈# a ∧ Neg p ∈# b ∧
    ((∃χ∈fst (N ∪ {C + {#Pos P#} + {#Neg P#}}, already-used).
      subsumes χ (a - {#Pos p#} + (b - {#Neg p#})))
    ∨ tautology (a - {#Pos p#} + (b - {#Neg p#})))
  by fastforce
moreover have tautology (C + {#Pos P#} + {#Neg P#}) by auto
ultimately show
  ∃p. Pos p ∈# a ∧ Neg p ∈# b
  ∧ ((∃χ∈fst (N - {C + {#Pos P#} + {#Neg P#}}, already-used).
    subsumes χ (a - {#Pos p#} + (b - {#Neg p#})))
  ∨ tautology (a - {#Pos p#} + (b - {#Neg p#})))
  by (metis (no-types) Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology
    sup-bot.right-neutral)
qed
qed

```

lemma

factoring-satisfiable: $I \models \{\#L\# \} + \{\#L\# \} + C \longleftrightarrow I \models \{\#L\# \} + C$ and
 resolution-satisfiable:
 consistent-interp $I \implies I \models \{\#Pos p\# \} + C \implies I \models \{\#Neg p\# \} + D \implies I \models C + D$ and
 factoring-same-vars: $\text{atms-of } (\{\#L\# \} + \{\#L\# \} + C) = \text{atms-of } (\{\#L\# \} + C)$
 unfolding true-cls-def consistent-interp-def by (fastforce split: if-split-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 $N\ N' :: 'v\ clauses$
assumes *inference-clause* $T\ T'$
and *total-over-m* $I\ (fst\ T)$
and *consistent: consistent-interp* I
shows $I \models_s fst\ T \longleftrightarrow I \models_s fst\ T \cup \{fst\ T'\}$
using *assms* **apply** (*induct rule: inference-clause.induct*)
unfolding *consistent-interp-def true-clss-def* **by** *auto force+*

lemma *inference-preserves-un-sat*:
fixes $N\ N' :: 'v\ clauses$
assumes *inference* $T\ T'$
and *total-over-m* $I\ (fst\ T)$
and *consistent: consistent-interp* I
shows $I \models_s fst\ T \longleftrightarrow I \models_s fst\ T'$
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 *atms-of-ms* $(fst\ (fst\ S \cup \{fst\ S'\}, snd\ S')) \subseteq atms-of-ms\ (fst\ S)$
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 $N\ N' :: 'v\ clauses$
assumes *inference* $T\ T'$
shows *atms-of-ms* $(fst\ T') \subseteq atms-of-ms\ (fst\ T)$
using *assms* **apply** (*induct rule: inference.induct*)
using *inference-clause-preserves-atms-of-ms* **by** *fastforce*

lemma *inference-preserves-total*:
fixes $N\ N' :: 'v\ clauses$
assumes *inference* $(N, already-used)\ (N', already-used')$
shows *total-over-m* $I\ N \implies total-over-m\ I\ N'$
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 $I \models_s \text{fst } N \longleftrightarrow I \models_s \text{fst } N'$
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply (*simp add*: *inference-preserves-un-sat*)
using *inference-preserves-un-sat rtranclp-inference-preserves-total* **by** *blast*

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

lemma *inference-preserves-finite-snd*:
assumes *inference* $\psi \ \psi'$ **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* $\psi \ \psi'$ **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* $P \notin \text{atm-of } I$
shows *consistent-interp* (*insert* $P \ I$)
proof –
have $P: \text{insert } P \ I = I \cup \{P\}$ **by** *auto*
show *?thesis* **unfolding** P
apply (*rule consistent-interp-disjoint*)
using *assms* **by** (*auto simp: image-iff*)
qed

lemma *simplify-clause-preserves-sat*:
assumes *simp: simplify* $\psi \ \psi'$
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**
cons: *consistent-interp I* **and**
tot: *total-over-m I ? \psi'*
using *sat* **unfolding** *satisfiable-def* **by** *auto*


```

{ assume Pos  $P \in I \vee \text{Neg } P \in I$ 
  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:  $\text{Pos } P \notin I$  and Neg:  $\text{Neg } P \notin I$ 
  then have consistent-interp ( $I \cup \{\text{Pos } P\}$ ) using cons by simp
  moreover have  $I'A: I \cup \{\text{Pos } P\} \models ?A'$  by auto
  have  $\{\text{Pos } P\} \cup I \models_s \psi - \{A + \{\# \text{Pos } P\}\} + \{\# \text{Neg } P\}\}$ 
    using  $\langle I \models_s \psi - \{A + \{\# \text{Pos } P\}\} + \{\# \text{Neg } P\}\rangle$  true-clss-union-increase' by blast
  then have  $I \cup \{\text{Pos } P\} \models_s \psi$ 
    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  $LL :: \text{'a literal multiset set} \Rightarrow \text{'a literal set}$  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 cons: consistent-interp  $I$  and 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: rtrancpl-induct)
apply simp-all
using simplify-preserves-unsat by blast

```

```

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

```

```

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

```

```

lemma sem-tree-size[case-names bigger]:
( $\bigwedge xs:: 'v \text{ sem-tree. } (\bigwedge ys:: 'v \text{ sem-tree. } \text{sem-tree-size } ys < \text{sem-tree-size } xs \Rightarrow P \text{ } ys) \Rightarrow P \text{ } xs$ )
 $\Rightarrow P \text{ } xs$ 
by (fact Nat.measure-induct-rule)

```

```

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

```

```

lemma simplify-preserve-partial-leaf:
simplify N N'  $\Rightarrow$  partial-interps Leaf I N  $\Rightarrow$  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 rtrancpl-inference-preserve-partial-tree:
assumes rtrancpl inference N N'
and partial-interps t I (fst N)
shows partial-interps t I (fst N')
using assms apply (induct rule: rtrancpl-induct, auto)
using inference-preserve-partial-tree by force

```

```

function build-sem-tree :: 'v :: linorder set  $\Rightarrow$  'v clauses  $\Rightarrow$  'v sem-tree where
build-sem-tree atms  $\psi$  =
  (if atms = {}  $\vee$   $\neg$  finite atms
   then Leaf
   else Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ ))
by auto
termination
  apply (relation measure ( $\lambda(A, -).$  card A), simp-all)
  apply (metis Min-in card-Diff1-less remove-def)+
done
declare build-sem-tree.induct[case-names tree]

lemma unsatisfiable-empty[simp]:
   $\neg$ unsatisfiable {}
  unfolding satisfiable-def apply auto
  using consistent-interp-def unfolding total-over-m-def total-over-set-def atms-of-ms-def by blast

lemma partial-interps-build-sem-tree-atms-general:
  fixes  $\psi :: 'v :: linorder$  clauses and  $p :: 'v$  literal list
  assumes unsat: unsatisfiable  $\psi$  and finite  $\psi$  and consistent-interp I
  and finite atms
  and atms-of-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:  $\bigwedge \chi. \chi \in \psi \implies$  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  $\neq$  {}
  have build-sem-tree atms  $\psi$  = Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    using build-sem-tree.simps[of atms  $\psi$ ] f atms by metis

  have consistent-interp (Ia  $\cup$  {Pos (Min atms)}) unfolding consistent-interp-def
    by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
      f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
      uminus-Neg uminus-Pos)

```

```

moreover have atms-of-ms  $\psi = \text{Set.remove } (\text{Min } \text{atms}) \text{ atms} \cup \text{atms-of-s } (Ia \cup \{\text{Pos } (\text{Min } \text{atms})\})$ 
  using Min-in atms f un by fastforce
moreover have disj':  $\text{Set.remove } (\text{Min } \text{atms}) \text{ atms} \cap \text{atms-of-s } (Ia \cup \{\text{Pos } (\text{Min } \text{atms})\}) = \{\}$ 
  by simp (metis disj disjoint-iff-not-equal member-remove)
moreover have finite ( $\text{Set.remove } (\text{Min } \text{atms}) \text{ atms}$ ) using f by (simp add: remove-def)
ultimately have subtree1: partial-interps (build-sem-tree ( $\text{Set.remove } (\text{Min } \text{atms}) \text{ atms}$ )  $\psi$ )
  ( $Ia \cup \{\text{Pos } (\text{Min } \text{atms})\}$ )  $\psi$ 
  using IH1[of  $Ia \cup \{\text{Pos } (\text{Min } (\text{atms}))\}$ ] atms f unsat finite by metis

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

moreover have disj':  $\text{Set.remove } (\text{Min } \text{atms}) \text{ atms} \cap \text{atms-of-s } (Ia \cup \{\text{Neg } (\text{Min } \text{atms})\}) = \{\}$ 
  using disj by auto
moreover have finite ( $\text{Set.remove } (\text{Min } \text{atms}) \text{ atms}$ ) using f by (simp add: remove-def)
ultimately have subtree2: partial-interps (build-sem-tree ( $\text{Set.remove } (\text{Min } \text{atms}) \text{ atms}$ )  $\psi$ )
  ( $Ia \cup \{\text{Neg } (\text{Min } \text{atms})\}$ )  $\psi$ 
  using IH2[of  $Ia \cup \{\text{Neg } (\text{Min } (\text{atms}))\}$ ] 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  $\psi = \text{atms-of-ms } \psi \cup \text{atms-of-s } \{\}$  unfolding atms-of-s-def by auto
  moreover have atms-of-ms  $\psi \cap \text{atms-of-s } \{\} = \{\}$  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 add: not-in-iff[symmetric])
next
case (Suc n χ)
note IH = this(1) and count = this(2) and L = this(3) and χ = this(4)
{
  assume n = 0
  then have inference** ψ ψ
  and χ ∈ fst ψ
  and ∀ L. (L ∈# χ) ⟷ (L ∈# χ)
  and count χ L = (1::nat)
  and ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ
    by (auto simp add: count L χ)
  then have ?case by metis
}
moreover {
  assume n > 0
  then have ∃ C. χ = C + {#L, L#}
    by (smt L Suc-eq-plus1-left add.left-commute add-diff-cancel-left' add-diff-cancel-right'
        count-greater-zero-iff count-single local.count multi-member-split plus-multiset.rep-eq)
  then obtain C where C: χ = C + {#L, L#} by metis
  let ?χ' = C + {#L#}
  let ?ψ' = (fst ψ ∪ {?χ'}, snd ψ)
  have φ: ∀ φ ∈ fst ψ. (φ ∈ fst ψ ∨ φ ≠ ?χ') ⟷ φ ∈ fst ?ψ' unfolding C by auto
  have inf: inference ψ ?ψ'
    using C factoring χ prod.collapse union-commute inference-step by metis
  moreover have count': count ?χ' L = n using C count by auto
  moreover have Lχ': L ∈# ?χ' by auto
  moreover have χ'ψ': ?χ' ∈ fst ?ψ' by auto
  ultimately obtain ψ'' and χ''
  where
    inference** ?ψ' ψ'' and
    α: χ'' ∈ fst ψ'' and
    ∀ La. (La ∈# ?χ') ⟷ (La ∈# χ'') and
    β: count χ'' L = (1::nat) and
    φ': ∀ φ. φ ∈ fst ?ψ' ⟶ φ ∈ fst ψ'' and
    Iχ: I ⊨ ?χ' ⟷ I ⊨ χ'' and
    tot: ∀ I'. total-over-m I' {?χ'} ⟶ total-over-m I' {χ''}
    using IH[of ?χ' ?ψ'] count' Lχ' χ'ψ' by blast

  then have inference** ψ ψ''
  and ∀ La. (La ∈# χ) ⟷ (La ∈# χ'')
  using inf unfolding C by auto
  moreover have ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ'' using φ φ' by metis
  moreover have I ⊨ χ ⟷ I ⊨ χ'' using Iχ unfolding true-cls-def C by auto
  moreover have ∀ I'. total-over-m I' {χ} ⟶ total-over-m I' {χ''}
    using tot unfolding C total-over-m-def by auto
  ultimately have ?case using φ φ' α β by metis
}
ultimately show ?case by auto
qed

```

lemma *can-decrease-tree-size:*

fixes ψ :: 'v state and tree :: 'v sem-tree
 assumes finite (fst ψ) and already-used-inv ψ

```

and partial-interps tree I (fst  $\psi$ )
shows  $\exists (tree': 'v \text{ sem-tree}) \psi'. \text{inference}^{**} \psi \psi' \wedge \text{partial-interps tree}' I (\text{fst } \psi')$ 
 $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$ 
using assms
proof (induct arbitrary: I rule: sem-tree-size)
case (bigger xs I) note IH = this(1) and finite = this(2) and a-u-i = this(3) and part = this(4)

{
  assume sem-tree-size xs = 0
  then have ?case using part by blast
}

moreover {
  assume sn0: sem-tree-size xs > 0
  obtain ag ad v where xs: xs = Node v ag ad using sn0 by (cases xs, auto)
  {
    assume sem-tree-size ag = 0 and sem-tree-size ad = 0
    then have ag: ag = Leaf and ad: ad = Leaf by (cases ag, auto) (cases ad, auto)

    then obtain  $\chi \chi'$  where
       $\chi: \neg I \cup \{Pos\ v\} \models \chi$  and
       $\text{tot}\chi: \text{total-over-m } (I \cup \{Pos\ v\}) \{\chi\}$  and
       $\chi\psi: \chi \in \text{fst } \psi$  and
       $\chi': \neg I \cup \{Neg\ v\} \models \chi'$  and
       $\text{tot}\chi': \text{total-over-m } (I \cup \{Neg\ v\}) \{\chi'\}$  and
       $\chi'\psi: \chi' \in \text{fst } \psi$ 
    using part unfolding xs by auto
    have Posv:  $\neg Pos\ v \in \# \chi$  using  $\chi$  unfolding true-cls-def true-lit-def by auto
    have Negv:  $\neg Neg\ v \in \# \chi'$  using  $\chi'$  unfolding true-cls-def true-lit-def by auto
    {
      assume Neg $\chi$ :  $\neg Neg\ v \in \# \chi$ 
      have  $\neg I \models \chi$  using  $\chi$  Posv unfolding true-cls-def true-lit-def by auto
      moreover have 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: rtrancp inference  $\psi \psi'$  and  $\chi_2\text{incl}: \chi_2 \in \text{fst } \psi'$ 
    and  $\chi\chi_2\text{-incl}: \forall L. L \in \# \chi \longleftrightarrow L \in \# \chi_2$ 
  }
}

```

```

and count $\chi 2$ : count  $\chi 2$  (Neg  $v$ ) = 1
and  $\varphi$ :  $\forall \varphi::'v$  literal multiset.  $\varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi'$ 
and  $I\chi$ :  $I \models \chi \longleftrightarrow I \models \chi 2$ 
and tot-imp $\chi$ :  $\forall I'$ . total-over-m  $I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi 2\}$ 
using can-decrease-count[of  $\chi$  Neg  $v$  count  $\chi$  (Neg  $v$ )  $\psi$   $I$ ]  $\chi\psi \chi'\psi$  by auto

have  $\chi' \in \text{fst } \psi'$  by (simp add:  $\chi'\psi \varphi$ )
with pos
obtain  $\psi'' \chi 2'$  where
  inf': inference**  $\psi' \psi''$ 
and  $\chi 2'$ -incl:  $\chi 2' \in \text{fst } \psi''$ 
and  $\chi'\chi 2$ -incl:  $\forall L::'v$  literal.  $(L \in \# \chi') = (L \in \# \chi 2')$ 
and count $\chi 2'$ : count  $\chi 2'$  (Pos  $v$ ) = (1::nat)
and  $\varphi'$ :  $\forall \varphi::'v$  literal multiset.  $\varphi \in \text{fst } \psi' \longrightarrow \varphi \in \text{fst } \psi''$ 
and  $I\chi'$ :  $I \models \chi' \longleftrightarrow I \models \chi 2'$ 
and tot-imp $\chi'$ :  $\forall I'$ . total-over-m  $I' \{\chi'\} \longrightarrow \text{total-over-m } I' \{\chi 2'\}$ 
using can-decrease-count[of  $\chi'$  Pos  $v$  count  $\chi'$  (Pos  $v$ )  $\psi' I$ ] by auto

obtain  $C$  where  $\chi 2$ :  $\chi 2 = C + \{\# \text{Neg } v \#\}$  and neg $C$ : Neg  $v \notin \# C$  and pos $C$ : Pos  $v \notin \# C$ 
proof –
  have  $\bigwedge m$ . Suc 0 – count  $m$  (Neg  $v$ ) = count ( $\chi 2 - m$ ) (Neg  $v$ )
    by (simp add: count $\chi 2$ )
  then show ?thesis
    using that by (metis (no-types) One-nat-def Posv Suc-inject Suc-pred  $\chi\chi 2$ -incl
      count-diff count-single insert-DiffM2 mem-Collect-eq multi-member-skip neg
      not-gr0 set-mset-def union-commute)
qed

obtain  $C'$  where
   $\chi 2'$ :  $\chi 2' = C' + \{\# \text{Pos } v \#\}$  and
  pos $C'$ : Pos  $v \notin \# C'$  and
  neg $C'$ : Neg  $v \notin \# C'$ 
proof –
  assume a1:  $\bigwedge C'$ .  $\llbracket \chi 2' = C' + \{\# \text{Pos } v \#\}; \text{Pos } v \notin \# C'; \text{Neg } v \notin \# C' \rrbracket \Longrightarrow \text{thesis}$ 
  have f2:  $\bigwedge n$ . (n::nat) – n = 0
    by simp
  have Neg  $v \notin \# \chi 2' - \{\# \text{Pos } v \#\}$ 
    using Negv  $\chi'\chi 2$ -incl by (auto simp: not-in-iff)
  have count  $\{\# \text{Pos } v \#\}$  (Pos  $v$ ) = 1
    by simp
  then show ?thesis
    by (metis  $\chi'\chi 2$ -incl  $\langle \text{Neg } v \notin \# \chi 2' - \{\# \text{Pos } v \#\} \rangle$  a1 count $\chi 2'$  count-diff f2
      insert-DiffM2 less-numeral-extra(3) mem-Collect-eq pos set-mset-def)
qed

have already-used-inv  $\psi'$ 
  using rtranclp-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 rtranclp-inference-preserves-already-used-inv a-u-i inf' unfolding tautology-def
  by simp

have tot $C$ : total-over-m  $I \{C\}$ 
  using tot-imp $\chi$  tot $\chi$  tot-over-m-remove[of  $I$  Pos  $v$   $C$ ] neg $C$  pos $C$  unfolding  $\chi 2$ 
  by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
have tot $C'$ : total-over-m  $I \{C'\}$ 

```

```

    using tot-imp $\chi'$  tot $\chi'$  total-over-m-sum tot-over-m-remove[of I Neg v C'] negC' posC'
    unfolding  $\chi 2'$  by (metis total-over-m-sum uminus-Neg)
  have  $\neg I \models C + C'$ 
    using  $\chi I\chi \chi' I\chi'$  unfolding  $\chi 2 \chi 2'$  true-cls-def by auto
  then have part-I- $\psi'''$ : partial-interps Leaf I (fst  $\psi'' \cup \{C + C'\}$ )
    using totC totC' by simp
    (metis  $\neg I \models C + C'$  atms-of-ms-singleton total-over-m-def total-over-m-sum)
}
{
  assume ( $\{\#Pos v\# \} + C', \{\#Neg v\# \} + C) \notin \text{snd } \psi''$ 
  then have inf'': inference  $\psi''$  (fst  $\psi'' \cup \{C + C'\}$ , snd  $\psi'' \cup \{(\chi 2', \chi 2)\}$ )
    using add.commute  $\varphi' \chi 2 \text{incl } \chi 2' \in \text{fst } \psi''$  unfolding  $\chi 2 \chi 2'$ 
    by (metis prod.collapse inference-step resolution)
  have inference**  $\psi$  (fst  $\psi'' \cup \{C + C'\}$ , snd  $\psi'' \cup \{(\chi 2', \chi 2)\}$ )
    using inf inf' inf'' rtranclp-trans by auto
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case using part-I- $\psi'''$  by (metis fst-conv)
}
moreover {
  assume a: ( $\{\#Pos v\# \} + C', \{\#Neg v\# \} + C) \in \text{snd } \psi''$ 
  then have ( $\exists \chi \in \text{fst } \psi''. (\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\chi\})$ 
     $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
     $\vee \text{tautology } (C' + C)$ )
  proof -
    obtain p where p: Pos p  $\in \# (\{\#Pos v\# \} + C')$  and
    n: Neg p  $\in \# (\{\#Neg v\# \} + C)$  and
    decomp: ( $(\exists \chi \in \text{fst } \psi''.$ 
      ( $\forall I. \text{total-over-m } I \{(\{\#Pos v\# \} + C') - \{\#Pos p\# \}$ 
         $+ ((\{\#Neg v\# \} + C) - \{\#Neg p\# \})\}$ 
         $\longrightarrow \text{total-over-m } I \{\chi\})$ 
       $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi$ 
         $\longrightarrow I \models (\{\#Pos v\# \} + C') - \{\#Pos p\# \} + ((\{\#Neg v\# \} + C) - \{\#Neg p\# \}))$ 
       $\vee \text{tautology } ((\{\#Pos v\# \} + C') - \{\#Pos p\# \} + ((\{\#Neg v\# \} + C) - \{\#Neg p\# \})))$ )
    using a by (blast intro: allE[OF a-u-i- $\psi''$ [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 n by force
    then have ?thesis unfolding Bex-def by auto
  }
  moreover {
    assume p = v
    then have ?thesis using decomp by (metis add.commute add-diff-cancel-left')
  }
  ultimately show ?thesis by auto
}
qed
moreover {
  assume  $\exists \chi \in \text{fst } \psi''. (\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\chi\})$ 
     $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
  then obtain  $\vartheta$  where  $\vartheta$ :  $\vartheta \in \text{fst } \psi''$  and
    tot- $\vartheta$ -CC':  $\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\vartheta\}$  and
     $\vartheta$ -inv:  $\forall I. \text{total-over-m } I \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$  by blast
  have partial-interps Leaf I (fst  $\psi''$ )
    using tot- $\vartheta$ -CC'  $\vartheta$   $\vartheta$ -inv totC totC'  $\neg I \models C + C'$  total-over-m-sum by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case by (metis inf inf' rtranclp-trans)
}

```



```

}
moreover {
  assume tautCC': tautology (C' + C)
  have total-over-m I {C'+C} using totC totC' total-over-m-sum by auto
  then have  $\neg$ 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 rtranclp.rtrancl-refl a-u-i by blast

  have partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi'$ )

```

```

    using rtrancplp-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 $\psi :: 'v :: \text{linorder state}$

assumes

unsat: $\neg \text{satisfiable (fst } \psi)$ **and**

finite: *finite* (fst ψ) **and**

a-u-v: *already-used-inv* ψ

shows $\exists \psi'. (\text{inference}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

proof –

obtain *tree* **where** *partial-interps* *tree* $\{\}$ (fst ψ)

using *partial-interps-build-sem-tree-atms* *assms* **by** *metis*

then show ?thesis

using *unsat* *finite* *a-u-v*

proof (*induct tree arbitrary*: ψ *rule*: *sem-tree-size*)

case (*bigger tree* ψ) **note** *H* = *this*

{

fix χ

assume *tree*: *tree* = *Leaf*

obtain χ **where** $\chi: \neg \{\} \models \chi$ **and** *tot* χ : *total-over-m* $\{\} \{\chi\}$ **and** $\chi\psi: \chi \in \text{fst } \psi$

using *H* **unfolding** *tree* **by** *auto*

moreover have $\{\#\} = \chi$

using *tot* χ **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' ψ' **where** *inf*: *inference*^{**} $\psi \psi'$ **and**

part': *partial-interps* *tree'* $\{\}$ (fst ψ') **and**

decrease: *sem-tree-size* *tree'* < *sem-tree-size* *tree* \vee *sem-tree-size* *tree* = 0

using *can-decrease-tree-size*[of ψ] *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 ψ') **using** *rtrancplp-inference-preserves-finite* *inf* *H*(4) **by** *metis*

moreover have *unsatisfiable* (fst ψ')

using *inference-preserves-unsat* *inf* *bigger.prem*s(2) **by** *blast*

moreover have *already-used-inv* ψ'

using *H*(5) *inf* *rtrancplp-inference-preserves-already-used-inv*[of $\psi \psi'$] **by** *auto*

ultimately have ?case **using** *inf* *rtrancplp-trans* *part'* *H*(1) **by** *fastforce*

}

ultimately show ?case **by** (*cases* *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  $\exists \psi'. (rtranclp\ inference\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')$ 
proof –
  have already-used-inv  $\psi$  unfolding assms by auto
  then show ?thesis using assms inference-completeness-inv by blast
qed

```

```

lemma inference-soundness:
  fixes  $\psi :: 'v :: linorder\ state$ 
  assumes rtranclp inference  $\psi\ \psi'$  and  $\{\#\} \in fst\ \psi'$ 
  shows unsatisfiable (fst  $\psi$ )
  using assms by (meson rtranclp-inference-preserves-un-sat satisfiable-def true-cls-empty true-cls-def)

```

```

lemma inference-soundness-and-completeness:
  fixes  $\psi :: 'v :: linorder\ state$ 
  assumes finite: finite (fst  $\psi$ )
  and snd  $\psi = \{\}$ 
  shows  $(\exists \psi'. (inference^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow \text{unsatisfiable } (fst\ \psi)$ 
  using assms inference-completeness inference-soundness by metis

```

12.4 Lemma about the simplified state

abbreviation *simplified* $\psi \equiv (no\text{-}step\ simplify\ \psi)$

```

lemma simplified-count:
  assumes simp: simplified  $\psi$  and  $\chi: \chi \in \psi$ 
  shows count  $\chi\ L \leq 1$ 
proof –
  {
    let  $? \chi' = \chi - \{\#L, L\#\}$ 
    assume count  $\chi\ L \geq 2$ 
    then have f1: count  $(\chi - \{\#L, L\#\} + \{\#L, L\\})\ L = \text{count } \chi\ L$ 
      by simp
    then have  $L \in \# \chi - \{\#L\#\}$ 
      by (metis (no-types) add.left-neutral add-diff-cancel-left' count-union diff-diff-add
        diff-single-trivial insert-DiffM mem-Collect-eq multi-member-this not-gr0 set-mset-def)
    then have  $\chi': ? \chi' + \{\#L\#\} + \{\#L\#\} = \chi$ 
      using f1 by (metis diff-diff-add diff-single-eq-union in-diffD)

    have  $\exists \psi'. \text{simplify } \psi\ \psi'$ 
      by (metis (no-types, hide-lams)  $\chi\ \chi'$  add commute factoring-imp-simplify union-assoc)
    then have False using simp by auto
  }
  then show ?thesis by arith
qed

```

```

lemma simplified-no-both:
  assumes simp: simplified  $\psi$  and  $\chi: \chi \in \psi$ 
  shows  $\neg (L \in \# \chi \wedge -L \in \# \chi)$ 
proof (rule ccontr)
  assume  $\neg \neg (L \in \# \chi \wedge -L \in \# \chi)$ 
  then have  $L \in \# \chi \wedge -L \in \# \chi$  by metis
  then obtain  $\chi'$  where  $\chi = \chi' + \{\#Pos\ (atm\text{-}of\ L)\#\} + \{\#Neg\ (atm\text{-}of\ L)\#\}$ 

```

```

    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  $\exists l'. \text{simplify } \{\psi\} \ l'$ 
    proof (induction rule: simplify.induct)
      case (tautology-deletion  $A\ P$ )
      have  $\{\#Neg\ P\# \} + (\{\#Pos\ P\# \} + (A + \{\#l\#\})) \in \{\psi\}$ 
        by (metis (no-types)  $A$  add.commute tautology-deletion.hyps union-lcomm)
      then show ?thesis
        by (metis simplify.tautology-deletion[of  $A + \{\#l\#\}$   $P\ \{\psi\}$ ] add.commute)
    next
      case (condensation  $A\ L$ )
      have  $A + \{\#L\#\} + \{\#L\#\} + \{\#l\#\} \in \{\psi\}$ 
        using  $A$  condensation.hyps by blast
      then have  $\{\#L, L\#\} + (A + \{\#l\#\}) \in \{\psi\}$ 
        by (metis (no-types) union-assoc union-commute)
      then show ?case
        using factoring-imp-simplify by blast
    next
      case (subsumption  $A\ B$ )
      then show ?case by blast
    qed
  }
  then have False using assms(1) by blast
}
ultimately show False by auto
qed

```

```

lemma in-simplified-simplified:
  assumes simp: simplified  $\psi$  and incl:  $\psi' \subseteq \psi$ 
  shows simplified  $\psi'$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $\psi''$  where simplify  $\psi' \psi''$  by metis
  then have  $\exists l'. \text{simplify } \psi l'$ 
  proof (induction rule: simplify.induct)
    case (tautology-deletion A P)
    then show ?thesis using simplify.tautology-deletion[of A P  $\psi$ ] incl by blast
  next
    case (condensation A L)
    then show ?case using simplify.condensation[of A L  $\psi$ ] incl by blast
  next
    case (subsumption A B)
    then show ?case using simplify.subsumption[of A  $\psi$  B] incl by auto
  qed
  then show False using assms(1) by blast
qed

lemma simplified-in:
  assumes simplified  $\psi$ 
  and  $N \in \psi$ 
  shows simplified  $\{N\}$ 
  using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

lemma subsumes-imp-formula:
  assumes  $\psi \leq \# \varphi$ 
  shows  $\{\psi\} \models_p \varphi$ 
  unfolding true-clss-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 (meson count-greater-eq-one-iff le-antisym simp simplified-count)
    then show False by (metis Suc-1  $\langle \chi \in \psi' \rangle$  not-less-eq-eq simp simplified-count)
  qed
qed

lemma simplified-no-more-full1-simplified:
  assumes simplified  $\psi$ 
  shows  $\neg \text{full1 simplify } \psi \psi'$ 
  using assms unfolding full1-def by (meson tranclpD)

```

12.5 Resolution and Invariants

inductive *resolution* :: 'v state \Rightarrow 'v state \Rightarrow bool **where**
full1-simp: *full1 simplify* $N\ N' \Rightarrow \text{resolution } (N, \text{already-used})\ (N', \text{already-used})$ |
inferring: *inference* $(N, \text{already-used})\ (N', \text{already-used}') \Rightarrow \text{simplified } N$
 $\Rightarrow \text{full simplify } N'\ N'' \Rightarrow \text{resolution } (N, \text{already-used})\ (N'', \text{already-used}')$

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 rtranclp-simplify-preserves-finite*
dest: *tranclp-into-rtranclp inference-preserves-finite*)

lemma *rtranclp-resolution-finite*:
assumes *resolution*** $\psi\ \psi'$ **and** *finite* (fst ψ)
shows *finite* (fst ψ')
using *assms* **by** (*induct rule*: *rtranclp-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 *rtranclp-resolution-finite-snd*:
assumes *resolution*** $\psi\ \psi'$ **and** *finite* (snd ψ)
shows *finite* (snd ψ')
using *assms* **by** (*induct rule*: *rtranclp-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 *tranclp-resolution-always-simplified*:
assumes *tranclp resolution* $\psi\ \psi'$
shows *simplified* (fst ψ')
using *assms* **by** (*induct rule*: *tranclp.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*: *rtranclp-simplify-atms-of-ms tranclp-into-rtranclp full1-def*)
by (*metis* (*no-types*, *lifting*) *contra-subsetD fst-conv full-def*
inference-preserves-atms-of-ms rtranclp-simplify-atms-of-ms subsetI)

lemma *rtranclp-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*: *rtranclp-induct*)
using *resolution-atms-of rtranclp-resolution-finite* **by** *blast+*

lemma *resolution-include*:

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

proof –

have *finite'*: *finite* (*fst* ψ') **using** *local.finite res resolution-finite* **by** *blast*
have *simplified* (*fst* ψ') **using** *res finite' resolution-always-simplified* **by** *blast*
then have $\text{fst } \psi' \subseteq \text{simple-clss } (\text{atms-of-ms } (\text{fst } \psi'))$
using *simplified-in-simple-clss finite' simplified-imp-distinct-mset-tauto*[*of fst* ψ'] **by** *auto*
moreover have $\text{atms-of-ms } (\text{fst } \psi') \subseteq \text{atms-of-ms } (\text{fst } \psi)$
using *res finite resolution-atms-of*[*of* ψ ψ'] **by** *auto*
ultimately show *?thesis* **by** (*meson atms-of-ms-finite local.finite order.trans rev-finite-subset simple-clss-mono*)

qed

lemma *rtranclp-resolution-include*:

assumes *res*: *trancpl resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows $\text{fst } \psi' \subseteq \text{simple-clss } (\text{atms-of-ms } (\text{fst } \psi))$
using *assms apply* (*induct rule: trancpl.induct*)
apply (*simp add: resolution-include*)
by (*meson simple-clss-mono order-class.le-trans resolution-include rtranclp-resolution-atms-of rtranclp-resolution-finite trancpl-into-rtranclp*)

abbreviation *already-used-all-simple*

$:: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ **where**

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

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

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

assumes $\text{vars} \subseteq \text{vars}'$
shows *already-used-all-simple* *a* *vars* \implies *already-used-all-simple* *a* *vars'*
using *assms* **by** *fast*

lemma *inference-clause-preserves-already-used-all-simple*:

assumes *inference-clause* *S* *S'*
and *already-used-all-simple* (*snd* *S*) *vars*
and *simplified* (*fst* *S*)
and $\text{atms-of-ms } (\text{fst } S) \subseteq \text{vars}$
shows *already-used-all-simple* (*snd* ($\text{fst } S \cup \{\text{fst } S'\}$, *snd* *S'*)) *vars*
using *assms*

proof (*induct rule: inference-clause.induct*)

case (*factoring* *L* *C* *N* *already-used*)

then show *?case* **by** (*simp add: simplified-in factoring-imp-simplify*)

next

case (*resolution* *P* *C* *N* *D* *already-used*) **note** *H* = *this*

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

proof –

fix *A* *B* *v*

assume $(A, B) \in \text{snd } (\text{fst } (N, \text{already-used}))$

$\cup \{\text{fst } (C + D, \text{already-used} \cup \{(\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)\}),$
 $\text{snd } (C + D, \text{already-used} \cup \{(\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)\})\}$

then have $(A, B) \in \text{already-used} \vee (A, B) = (\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)$ **by** *auto*

moreover {

assume $(A, B) \in \text{already-used}$

then have $\text{simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars}$

```

    using  $H(4)$  by auto
  }
  moreover {
    assume eq:  $(A, B) = (\{\#Pos\ P\# \} + C, \{\#Neg\ P\# \} + D)$ 
    then have simplified  $\{A\}$  using simplified-in  $H(1,5)$  by auto
    moreover have simplified  $\{B\}$  using eq simplified-in  $H(2,5)$  by auto
    moreover have  $atms-of\ A \subseteq atms-of-ms\ N$ 
      using eq  $H(1)$ 
      using  $atms-of-atms-of-ms-mono[of\ A\ N]$  by auto
    moreover have  $atms-of\ B \subseteq atms-of-ms\ N$ 
      using eq  $H(2)$   $atms-of-atms-of-ms-mono[of\ B\ N]$  by auto
    ultimately have  $simplified\ \{A\} \wedge simplified\ \{B\} \wedge atms-of\ A \subseteq vars \wedge atms-of\ B \subseteq vars$ 
      using  $H(6)$  by auto
  }
  ultimately show  $simplified\ \{A\} \wedge simplified\ \{B\} \wedge atms-of\ A \subseteq vars \wedge atms-of\ B \subseteq vars$ 
    by fast
qed
qed

```

lemma *inference-preserves-already-used-all-simple:*

```

  assumes inference  $S\ S'$ 
  and already-used-all-simple  $(snd\ S)\ vars$ 
  and simplified  $(fst\ S)$ 
  and  $atms-of-ms\ (fst\ S) \subseteq vars$ 
  shows already-used-all-simple  $(snd\ S')\ vars$ 
  using assms
proof (induct rule: inference.induct)
  case (inference-step  $S$  clause already-used)
  then show ?case
    using inference-clause-preserves-already-used-all-simple[ $of\ S\ (clause, already-used)\ vars$ ]
    by auto
qed

```

lemma *already-used-all-simple-inv:*

```

  assumes resolution  $S\ S'$ 
  and already-used-all-simple  $(snd\ S)\ vars$ 
  and  $atms-of-ms\ (fst\ S) \subseteq vars$ 
  shows already-used-all-simple  $(snd\ S')\ vars$ 
  using assms
proof (induct rule: resolution.induct)
  case (full1-simp  $N\ N'$ )
  then show ?case by simp
next
  case (inferring  $N$  already-used  $N'$  already-used'  $N''$ )
  then show already-used-all-simple  $(snd\ (N'', already-used'))\ vars$ 
    using inference-preserves-already-used-all-simple[ $of\ (N, already-used)$ ] by simp
qed

```

lemma *rtrancp-already-used-all-simple-inv:*

```

  assumes resolution**  $S\ S'$ 
  and already-used-all-simple  $(snd\ S)\ vars$ 
  and  $atms-of-ms\ (fst\ S) \subseteq vars$ 
  and finite  $(fst\ S)$ 
  shows already-used-all-simple  $(snd\ S')\ vars$ 
  using assms

```


proof (*induct rule: rtrancpl-induct*)
 case *base*
 then show ?*case* **by** *simp*
next
 case (*step S' S''*) **note** *infstar = this(1)* **and** *IH = this(3)* **and** *res = this(2)* **and**
already = this(4) **and** *atms = this(5)* **and** *finite = this(6)*
have *already-used-all-simple (snd S')* **vars using** *IH already atms finite* **by** *simp*
moreover have *atms-of-ms (fst S') \subseteq atms-of-ms (fst S)*
by (*simp add: infstar local.finite rtrancpl-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 trancplD*)
by (*metis inference-simplified-already-used-subset fst-conv snd-conv*)

lemma *trancpl-resolution-simplified-already-used-subset:*
assumes *trancpl resolution S S'*
and *simplified (fst S)*
shows *snd S \subset snd S'*
using *assms* **apply** (*induct rule: trancpl.induct*)
using *resolution-simplified-already-used-subset* **apply** *metis*
by (*meson trancpl-resolution-always-simplified resolution-simplified-already-used-subset less-trans*)

abbreviation *already-used-top vars \equiv simple-clss vars \times simple-clss vars*

lemma *already-used-all-simple-in-already-used-top:*
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *s \subseteq already-used-top vars*
proof
 fix *x*
 assume *x-s: x \in s*
 obtain *A B* **where** *x: x = (A, B)* **by** (*cases x, auto*)
 then have *simplified {A}* **and** *atms-of A \subseteq vars* **using** *assms(1) x-s* **by** *fastforce+*

then have $A: A \in \text{simple-clss vars}$
using *simple-clss-mono*[of *atms-of* A *vars*] x *assms*(2)
simplified-imp-distinct-mset-tauto[of $\{A\}$]
distinct-mset-not-tautology-implies-in-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 $B: B \in \text{simple-clss vars}$
using *simplified-imp-distinct-mset-tauto*[of $\{B\}$]
distinct-mset-not-tautology-implies-in-simple-clss
simple-clss-mono[of *atms-of* B *vars*] x *assms*(2) **by** *fast*
ultimately show $x \in \text{simple-clss vars} \times \text{simple-clss vars}$
unfolding x **by** *auto*
qed

lemma *already-used-top-finite*:
assumes *finite vars*
shows *finite* (*already-used-top vars*)
using *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 simple-clss-mono* **by** *auto*

lemma *already-used-all-simple-finite*:
fixes $s :: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set}$ **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* $\psi \equiv \text{card} (\text{already-used-top vars} - \psi)$

lemma *resolution-card-simple-decreasing*:
assumes *res: resolution $\psi \psi'$*
and *a-u-s: already-used-all-simple (snd 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 $?top = \text{simple-clss } ?vars \times \text{simple-clss } ?vars$
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*]

$\text{card-mono}[\text{of already-used-top vars snd } \psi'] \text{ already-used-top-finite}[OF \text{ finite-}v] \text{ by metis}$
then show $?thesis$
using $\text{psubset-card-mono}[OF f \text{ resolution-simplified-already-used-subset}[OF \text{ res simp}]]$
unfolding 1 2 **by** linarith
qed

lemma $\text{trancp-resolution-card-simple-decreasing}$:

assumes $\text{trancp resolution } \psi \psi'$ **and** $\text{finite-fst: finite (fst } \psi)$
and $\text{already-used-all-simple (snd } \psi) \text{ vars}$
and $\text{atms-of-ms (fst } \psi) \subseteq \text{vars}$
and $\text{finite-v: finite vars}$
and $\text{finite-snd: finite (snd } \psi)$
and $\text{simplified (fst } \psi)$
shows $\text{card-simple vars (snd } \psi') < \text{card-simple vars (snd } \psi)$
using assms
proof ($\text{induct rule: trancp-induct}$)
case ($\text{base } \psi'$)
then show $?case$ **by** ($\text{simp add: resolution-card-simple-decreasing}$)
next
case ($\text{step } \psi' \psi''$) **note** $\text{res} = \text{this}(1)$ **and** $\text{res}' = \text{this}(2)$ **and** $\text{a-u-s} = \text{this}(5)$ **and**
 $\text{atms} = \text{this}(6)$ **and** $\text{f-v} = \text{this}(7)$ **and** $\text{f-fst} = \text{this}(4)$ **and** $H = \text{this}$
then have $\text{card-simple vars (snd } \psi') < \text{card-simple vars (snd } \psi)$ **by** auto
moreover have $\text{a-u-s': already-used-all-simple (snd } \psi') \text{ vars}$
using $\text{rtrancp-already-used-all-simple-inv}[OF \text{ trancp-into-rtrancp}[OF \text{ res}] \text{ a-u-s atms f-fst}]$.
have $\text{finite (fst } \psi')$
by ($\text{meson finite-fst res rtrancp-resolution-finite trancp-into-rtrancp}$)
moreover have $\text{finite (snd } \psi')$ **using** $\text{already-used-all-simple-finite}[OF \text{ a-u-s' f-v}]$.
moreover have $\text{simplified (fst } \psi')$ **using** $\text{res trancp-resolution-always-simplified}$ **by** blast
moreover have $\text{atms-of-ms (fst } \psi') \subseteq \text{vars}$
by ($\text{meson atms f-fst order.trans res rtrancp-resolution-atms-of trancp-into-rtrancp}$)
ultimately show $?case$
using $\text{resolution-card-simple-decreasing}[OF \text{ res' a-u-s' f-v}] \text{ f-v}$
 $\text{less-trans}[\text{of card-simple vars (snd } \psi'') \text{ card-simple vars (snd } \psi')$
 $\text{card-simple vars (snd } \psi)]$
by blast
qed

lemma $\text{trancp-resolution-card-simple-decreasing-2}$:

assumes $\text{trancp resolution } \psi \psi'$
and $\text{finite-fst: finite (fst } \psi)$
and $\text{empty-snd: snd } \psi = \{\}$
and $\text{simplified (fst } \psi)$
shows $\text{card-simple (atms-of-ms (fst } \psi)) (\text{snd } \psi') < \text{card-simple (atms-of-ms (fst } \psi)) (\text{snd } \psi)$
proof –
let $?vars = (\text{atms-of-ms (fst } \psi))$
have $\text{already-used-all-simple (snd } \psi) ?vars$ **unfolding** empty-snd **by** auto
moreover have $\text{atms-of-ms (fst } \psi) \subseteq ?vars$ **by** auto
moreover have $\text{finite-v: finite } ?vars$ **using** finite-fst **by** auto
moreover have $\text{finite-snd: finite (snd } \psi)$ **unfolding** empty-snd **by** auto
ultimately show $?thesis$
using $\text{assms}(1,2,4) \text{ trancp-resolution-card-simple-decreasing}[\text{of } \psi \psi']$ **by** presburger
qed

12.5.2 well-foundness if the relation

lemma *wf-simplified-resolution*:

assumes *f-vars*: *finite vars*

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

proof –

```

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

unfolding *wf-def*

apply (*simp add: resolution-always-simplified*)

by (*metis (mono-tags, hide-lams) fst-conv resolution-always-simplified*)

lemma *wf-resolution*:

assumes *f-vars*: *finite vars*

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

proof –

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

then show *wf (?R ∪ ?S)*

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

qed

lemma *rtrncp-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* $\psi' \longrightarrow \text{satisfiable } \psi$
using *assms* *rtranclp-simplify-preserves-unsat*[*of* $\psi \ \psi'$] **unfolding** *full-def* **by** *metis*

lemma *resolution-preserves-unsat*:

assumes *resolution* $\psi \ \psi'$
shows *satisfiable* $(fst \ \psi') \longrightarrow \text{satisfiable } (fst \ \psi)$
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 *rtranclp-resolution-preserves-unsat*:

assumes *resolution*** $\psi \ \psi'$
shows *satisfiable* $(fst \ \psi') \longrightarrow \text{satisfiable } (fst \ \psi)$
using *assms* **apply** *induction*
using *resolution-preserves-unsat* **by** *fast+*

lemma *rtranclp-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* *rtranclp-simplify-preserve-partial-tree*[*of* $N \ N' \ t \ I$] *tranclp-into-rtranclp*
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* *rtranclp-simplify-preserve-partial-tree*[*of* $N \ N' \ t \ I$] *tranclp-into-rtranclp*
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 *rtranclp-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 (rename-tac n, case-tac n) auto

lemma wf-always-more-step-False:
  assumes wf R
  shows  $(\forall x. \exists z. (z, x) \in R) \implies \text{False}$ 
  using assms unfolding wf-def by (meson Domain.DomainI assms wfE-min)

lemma finite-finite-mset-element-of-mset[simp]:
  assumes finite N
  shows finite  $\{f \ \varphi \ L \mid \varphi \ L. \ \varphi \in N \wedge L \in \# \ \varphi \wedge P \ \varphi \ L\}$ 
  using assms
proof (induction N rule: finite-induct)
  case empty
  show ?case by auto
next
  case (insert x N) note finite = this(1) and IH = this(3)
  have  $\{f \ \varphi \ L \mid \varphi \ L. (\varphi = x \vee \varphi \in N) \wedge L \in \# \ \varphi \wedge P \ \varphi \ L\} \subseteq \{f \ x \ L \mid L. L \in \# \ x \wedge P \ x \ L\}$ 
     $\cup \{f \ \varphi \ L \mid \varphi \ L. \varphi \in N \wedge L \in \# \ \varphi \wedge P \ \varphi \ L\}$  by auto
  moreover have finite  $\{f \ x \ L \mid L. L \in \# \ x\}$  by auto
  ultimately show ?case using IH finite-subset by fastforce
qed

value card
value filter-mset
value  $\{\# \text{count } \varphi \ L \mid L \in \# \ \varphi. \ 2 \leq \text{count } \varphi \ L \#\}$ 
value  $(\lambda \varphi. \text{msetsum } \{\# \text{count } \varphi \ L \mid L \in \# \ \varphi. \ 2 \leq \text{count } \varphi \ L \#\})$ 

syntax
  -comprehension1'-mset :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b multiset  $\Rightarrow$  'a multiset
    (( $\{\# \cdot / \cdot \cdot : \text{setof } \cdot \#\}$ )))
translations
   $\{\# e. x. \text{setof } M \#\} == \text{CONST set-mset } (\text{CONST image-mset } (\%x. e) \ M)$ 
value  $\{\# \ a. \ a : \text{setof } \{\# 1, 1, 2 :: \text{int}\} \#\} = \{1, 2\}$ 

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

interpretation sum-count-ge-2:
  folding  $(\lambda \varphi. \text{op } + (\text{msetsum } \{\# \text{count } \varphi \ L \mid L \in \# \ \varphi. \ 2 \leq \text{count } \varphi \ L \#\})) \ 0$ 
rewrites
  folding.F  $(\lambda \varphi. \text{op } + (\text{msetsum } \{\# \text{count } \varphi \ L \mid L \in \# \ \varphi. \ 2 \leq \text{count } \varphi \ L \#\})) \ 0 = \text{sum-count-ge-2}$ 
proof –
  show folding  $(\lambda \varphi. \text{op } + (\text{msetsum } (\text{image-mset } (\text{count } \varphi) \ \{\# \ L \in \# \ \varphi. \ 2 \leq \text{count } \varphi \ L \#\})))$ 
    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  $(\lambda \varphi. \text{op } + (\text{msetsum } (\text{image-mset } (\text{count } \varphi) \ \{\# \ L \in \# \ \varphi. \ 2 \leq \text{count } \varphi \ L \#\}))) \ 0$ 
    = sum-count-ge-2 by (auto simp add: sum-count-ge-2-def)
qed

```

```

lemma finite-incl-le-setsum:
  finite (B::'a 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  $\in A$ )
    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  $\in A$ 
  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  $\Rightarrow$  nat  $\Rightarrow$  nat where
         $\forall x0\ x1. (\exists v2. x0 = x1 + v2) = (x0 = x1 + nn\ x0\ x1)$ 
      by moura
      then have  $\Xi F = \Xi (A - \{a\}) + nn (\Xi F) (\Xi (A - \{a\}))$ 
      by (meson  $\langle \Xi (A - \{a\}) \leq \Xi F \rangle$  le-iff-add)
      then show ?thesis
        by (metis (no-types) 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 simplify-finite-measure-decrease:
  simplify N N'  $\implies$  finite N  $\implies$  card N' +  $\Xi N' <$  card N +  $\Xi N$ 
proof (induction rule: simplify.induct)
case (tautology-deletion A P) note an = this(1) and fin = this(2)
let ?N' = N - {A + {#Pos P#} + {#Neg P#}}
have card ?N' < card N
  by (meson card-Diff1-less tautology-deletion.hyps tautology-deletion.premis)
moreover have ?N'  $\subseteq N$  by auto
then have sum-count-ge-2 ?N'  $\leq$  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 = this(1) and fin = this(2)
let ?C' = A + {#L#}
let ?C = A + {#L#} + {#L#}
let ?N' = N - {?C}  $\cup$  {?C'}
have card ?N'  $\leq$  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:
       $\{\# L_a \in \# A. (L = L_a \longrightarrow L_a \in \# A) \wedge (L \neq L_a \longrightarrow 2 \leq \text{count } A\ L_a)\#\}$ 
       $= \{\# L_a \in \# A. L \neq L_a \wedge 2 \leq \text{count } A\ L_a\#\} +$ 
       $\{\# L_a \in \# A. L = L_a \wedge \text{Suc } 0 \leq \text{count } A\ L_a\#\}$ 
    by (auto simp: multiset-eq-iff ac-simps)

```



```

have mset-decomp2: {# La ∈# A. L ≠ La → 2 ≤ count A La#} =
  {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
  by (auto simp: multiset-eq-iff)
show ?thesis
  by (auto simp: mset-decomp mset-decomp2 filter-mset-eq ac-simps)
qed
have  $\exists N' < \exists N$ 
proof cases
  assume a1:  $?C' \in N$ 
  then show ?thesis
    proof -
      have f2:  $\bigwedge m M. \text{insert } (m::'a \text{ literal multiset}) (M - \{m\}) = M \cup \{m\} \vee m \notin M$ 
        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:  $\bigwedge M m. M - \{m::'a \text{ literal multiset}\} = M \cup \{m\} \vee m \in M$ 
        using Diff-insert-absorb Un-empty-right by fastforce
      have f5:  $\text{insert } (A + \{\#L\# \} + \{\#L\# \}) N = N$ 
        using f3 f2 Un-empty-right condensation.hyps insert-iff by fastforce
      have  $\bigwedge m M. \text{insert } (m::'a \text{ literal multiset}) M = M \cup \{m\} \vee m \notin M$ 
        using f3 f2 Un-empty-right add.right-neutral insert-iff by fastforce
      then have  $\exists (N - \{A + \{\#L\# \} + \{\#L\# \}) < \exists N$ 
        using f5 f4 by (metis Un-empty-right  $\langle \exists \{A + \{\#L\# \} \} < \exists \{A + \{\#L\# \} + \{\#L\# \} \rangle$ 
          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 ∈# A. (L = La → Suc 0 ≤ count A La) ∧ (L ≠ La → 2 ≤ count A La)#}
      = {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} +
        {# La ∈# A. L = La ∧ Suc 0 ≤ count A L#}
        by (auto simp: multiset-eq-iff ac-simps)
    have mset-decomp2: {# La ∈# A. L ≠ La → 2 ≤ count A La#} =
      {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
      by (auto simp: multiset-eq-iff)

    show ?thesis
      using  $\langle \exists \{A + \{\#L\# \} \} < \exists \{A + \{\#L\# \} + \{\#L\# \} \rangle$  condensation.hyps fin
        sum-count-ge-2.remove[of - A + {#L#} + {#L#}]  $\langle ?C' \notin N \rangle$ 
      by (auto simp: mset-decomp mset-decomp2 filter-mset-eq)
    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  $\exists (N - \{B\}) \leq \exists N$ 
  by (simp add: Diff-subset finite-incl-le-setsum subsumption.prem)
ultimately show ?case by linarith
qed
lemma simplify-terminates:

```

$wf \{(N', N). \text{finite } N \wedge \text{simplify } N N'\}$
using *assms* **apply** (rule *wfP-if-measure*[of *finite simplify* $\lambda N. \text{card } N + \Xi N$])
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 *rtranclp-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 rtranclp-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 *rtranclp-simplify-terminates*[*OF assms*] **unfolding** *full1-def*
by (*metis Nitpick.rtranclp-unfold*)

```

lemma finite-simplified-full-simp:
  assumes finite N
  shows  $\exists N'. \text{full simplify } N N'$ 
  using rtranchp-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 (cases xs, auto)
    {
      assume sem-tree-size ag = 0  $\wedge$  sem-tree-size ad = 0
      then have ag: ag = Leaf and ad: ad = Leaf by (cases ag, auto, cases ad, auto)

      then obtain  $\chi \chi'$  where
         $\chi: \neg I \cup \{\text{Pos } v\} \models \chi$  and
        tot $\chi$ : total-over-m ( $I \cup \{\text{Pos } v\}$ )  $\{\chi\}$  and
         $\chi\psi: \chi \in \text{fst } \psi$  and
         $\chi': \neg I \cup \{\text{Neg } v\} \models \chi'$  and
        tot $\chi'$ : total-over-m ( $I \cup \{\text{Neg } v\}$ )  $\{\chi'\}$  and  $\chi'\psi: \chi' \in \text{fst } \psi$ 
        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 $\chi: \neg \text{Neg } v \in \# \chi$ 
        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 $\chi: \neg \text{Pos } v \in \# \chi'$ 
      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 (simp add: dual-order.antisym)
  have count  $\chi'$  (Pos v) = 1
    using simplified-count[OF simp  $\chi'\psi$ ] pos
    by (simp add: dual-order.antisym)

  obtain C where  $\chi C$ :  $\chi = C + \{\#Neg\ v\# \}$  and negC: Neg v  $\notin\#$  C and posC: Pos v  $\notin\#$  C
  by (metis (no-types, lifting) One-nat-def Posv Suc-eq-plus1-left  $\langle$ count  $\chi$  (Neg v) = 1 $\rangle$ 
    add-diff-cancel-left' count-diff count-greater-eq-one-iff count-single insert-DiffM
    insert-DiffM2 less-numeral-extra(3) multi-member-skip not-le not-less-eq-eq)

  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, lifting) One-nat-def Negv Suc-eq-plus1-left  $\langle$ count  $\chi'$  (Pos v) = 1 $\rangle$ 
    add-diff-cancel-left' count-diff count-greater-eq-one-iff count-single insert-DiffM
    insert-DiffM2 less-numeral-extra(3) multi-member-skip not-le not-less-eq-eq)

  have totC: total-over-m I {C}
    using tot $\chi$  total-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 total-over-m-remove[of I Neg v C'] negC' posC'
    unfolding  $\chi C'$  by (metis total-over-m-sum uminus-Neg)
  have  $\neg I \models C + C'$ 
    using  $\chi$   $\chi'$   $\chi C$   $\chi C'$  by auto
  then have part-I- $\psi'''$ : partial-interps Leaf I (fst  $\psi \cup \{C + C'\}$ )
    using totC totC'  $\neg I \models C + C'$  by (metis Un-insert-right insertI1
    partial-interps.simps(1) total-over-m-sum)
  {
    assume ( $\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C$ )  $\notin$  snd  $\psi$ 
    then have inf'': inference  $\psi$  (fst  $\psi \cup \{C + C'\}$ , snd  $\psi \cup \{(\chi', \chi)\}$ )
      by (metis  $\chi'\psi$   $\chi C$   $\chi C'$   $\chi\psi$  add.commute inference-step prod.collapse resolution)
    obtain N' where full: full simplify (fst  $\psi \cup \{C + C'\}$ ) N'
      by (metis finite-simplified-full-simp fst-conv inf'' inference-preserves-finite
        local.finite)
    have resolution  $\psi$  (N', snd  $\psi \cup \{(\chi', \chi)\}$ )
      using resolution.intros(2)[OF - simp full, of snd  $\psi$  snd  $\psi \cup \{(\chi', \chi)\}$ ] inf''
      by (metis surjective-pairing)
    moreover have partial-interps Leaf I N'
      using full-simplify-preserve-partial-tree[OF full part-I- $\psi'''$ ] .
    moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
    ultimately have ?case
      by (metis (no-types) prod.sel(1) rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
  }
}
moreover {
  assume a: ( $\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C$ )  $\in$  snd  $\psi$ 

```

then have $(\exists \chi \in \text{fst } \psi. (\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\chi\})$
 $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)) \vee \text{tautology } (C' + C)$
proof –
obtain p **where** $p: \text{Pos } p \in \# (\{\# \text{Pos } v\# \} + C') \wedge \text{Neg } p \in \# (\{\# \text{Neg } v\# \} + C)$
 $\wedge ((\exists \chi \in \text{fst } \psi. (\forall I. \text{total-over-m } I \{(\{\# \text{Pos } v\# \} + C') - \{\# \text{Pos } p\# \} + ((\{\# \text{Neg } v\# \} + C)$
 $+ C) - \{\# \text{Neg } p\# \})) \longrightarrow \text{total-over-m } I \{\chi\}) \wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\# \text{Pos } v\# \} + C')$
 $- \{\# \text{Pos } p\# \} + ((\{\# \text{Neg } v\# \} + C) - \{\# \text{Neg } p\# \}))) \vee \text{tautology } ((\{\# \text{Pos } v\# \} + C') -$
 $\{\# \text{Pos } p\# \} + ((\{\# \text{Neg } v\# \} + C) - \{\# \text{Neg } p\# \})))$
using a **by** $(\text{blast intro: allE[OF a-u-i[unfolding subsumes-def Ball-def],$
 $\text{of } (\{\# \text{Pos } v\# \} + C', \{\# \text{Neg } v\# \} + C)])$
{ assume $p \neq v$
then have $\text{Pos } p \in \# C' \wedge \text{Neg } p \in \# C$ **using** p **by** force
then have $?thesis$ **by** auto
}
moreover {
assume $p = v$
then have $?thesis$ **using** p **by** $(\text{metis add.commute add-diff-cancel-left})$
}
ultimately show $?thesis$ **by** auto
qed
moreover {
assume $\exists \chi \in \text{fst } \psi. (\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\chi\})$
 $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$
then obtain ϑ **where**
 $\vartheta: \vartheta \in \text{fst } \psi$ **and**
 $\text{tot-}\vartheta\text{-}CC': \forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\vartheta\}$ **and**
 $\vartheta\text{-inv}: \forall I. \text{total-over-m } I \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$ **by** blast
have $\text{partial-interps Leaf } I (\text{fst } \psi)$
using $\text{tot-}\vartheta\text{-}CC' \vartheta \vartheta\text{-inv totC totC' } \hookrightarrow I \models C + C'$ **total-over-m-sum** **by** fastforce
moreover have $\text{sem-tree-size Leaf} < \text{sem-tree-size } xs$ **unfolding** xs **by** auto
ultimately have $?case$ **by** blast
}
moreover {
assume $\text{tautCC': tautology } (C' + C)$
have $\text{total-over-m } I \{C'+C\}$ **using** $\text{totC totC' total-over-m-sum}$ **by** auto
then have $\neg \text{tautology } (C' + C)$
using $\hookrightarrow I \models C + C'$ **unfolding** $\text{add.commute[of } C \ C']$ **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** $(\text{metis (no-types) sem-tree-size.simps(1)})$
}
moreover {
assume $\text{size-ag: sem-tree-size ag} > 0$
have $\text{sem-tree-size ag} < \text{sem-tree-size } xs$ **unfolding** xs **by** auto
moreover have $\text{partial-interps ag } (I \cup \{\text{Pos } v\}) (\text{fst } \psi)$
and $\text{partad: partial-interps ad } (I \cup \{\text{Neg } v\}) (\text{fst } \psi)$
using $\text{part partial-interps.simps(2)}$ **unfolding** xs **by** metis+
moreover
have $\text{sem-tree-size ag} < \text{sem-tree-size } xs \implies \text{finite } (\text{fst } \psi) \implies \text{already-used-inv } \psi$
 $\implies \text{partial-interps ag } (I \cup \{\text{Pos } v\}) (\text{fst } \psi) \implies \text{simplified } (\text{fst } \psi)$

```

     $\implies \exists \text{tree}' \psi'. \text{resolution}^{**} \psi \psi' \wedge \text{partial-interps tree}' (I \cup \{\text{Pos } v\}) (\text{fst } \psi')$ 
     $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size ag} \vee \text{sem-tree-size ag} = 0)$ 
    using IH[of ag  $I \cup \{\text{Pos } v\}$ ] by auto
  ultimately obtain  $\psi' :: 'v \text{ state and tree}' :: 'v \text{ sem-tree}$  where
    inf:  $\text{resolution}^{**} \psi \psi'$ 
    and part:  $\text{partial-interps tree}' (I \cup \{\text{Pos } v\}) (\text{fst } \psi')$ 
    and size:  $\text{sem-tree-size tree}' < \text{sem-tree-size ag} \vee \text{sem-tree-size ag} = 0$ 
    using finite part rtranclp.rtrancl-refl a-u-i simp by blast

  have  $\text{partial-interps ad} (I \cup \{\text{Neg } v\}) (\text{fst } \psi')$ 
    using rtranclp-resolution-preserve-partial-tree inf partad by fast
  then have  $\text{partial-interps (Node } v \text{ tree}' \text{ ad)} I (\text{fst } \psi')$  using part by auto
  then have ?case using inf size size-ag part unfolding xs by fastforce
}
moreover {
  assume size-ad:  $\text{sem-tree-size ad} > 0$ 
  have  $\text{sem-tree-size ad} < \text{sem-tree-size xs}$  unfolding xs by auto
  moreover
    have
      partag:  $\text{partial-interps ag} (I \cup \{\text{Pos } v\}) (\text{fst } \psi)$  and
       $\text{partial-interps ad} (I \cup \{\text{Neg } v\}) (\text{fst } \psi)$ 
      using part partial-interps.simps(2) unfolding xs by metis+
    moreover have  $\text{sem-tree-size ad} < \text{sem-tree-size xs} \longrightarrow \text{finite (fst } \psi) \longrightarrow \text{already-used-inv } \psi$ 
       $\longrightarrow (\text{partial-interps ad} (I \cup \{\text{Neg } v\}) (\text{fst } \psi) \longrightarrow \text{simplified (fst } \psi))$ 
       $\longrightarrow (\exists \text{tree}' \psi'. \text{resolution}^{**} \psi \psi' \wedge \text{partial-interps tree}' (I \cup \{\text{Neg } v\}) (\text{fst } \psi'))$ 
       $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size ad} \vee \text{sem-tree-size ad} = 0))$ 
      using IH by blast
    ultimately obtain  $\psi' :: 'v \text{ state and tree}' :: 'v \text{ sem-tree}$  where
      inf:  $\text{resolution}^{**} \psi \psi'$ 
      and part:  $\text{partial-interps tree}' (I \cup \{\text{Neg } v\}) (\text{fst } \psi')$ 
      and size:  $\text{sem-tree-size tree}' < \text{sem-tree-size ad} \vee \text{sem-tree-size ad} = 0$ 
      using finite part rtranclp.rtrancl-refl a-u-i simp by blast

    have  $\text{partial-interps ag} (I \cup \{\text{Pos } v\}) (\text{fst } \psi)$ 
      using rtranclp-resolution-preserve-partial-tree inf partag by fast
    then have  $\text{partial-interps (Node } v \text{ ag tree')} I (\text{fst } \psi')$  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 *resolution-completeness-inv*:

fixes $\psi :: 'v :: \text{linorder state}$

assumes

unsat: $\neg \text{satisfiable (fst } \psi)$ and

finite: $\text{finite (fst } \psi)$ and

a-u-v: $\text{already-used-inv } \psi$

shows $\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

proof –

obtain *tree* where $\text{partial-interps tree } \{\} (\text{fst } \psi)$

using *partial-interps-build-sem-tree-atms assms* by *metis*

then show ?thesis

using *unsat finite a-u-v*

```

proof (induct tree arbitrary:  $\psi$  rule: sem-tree-size)
  case (bigger tree  $\psi$ ) note  $H = \text{this}$ 
  {
    fix  $\chi$ 
    assume  $\text{tree: tree} = \text{Leaf}$ 
    obtain  $\chi$  where  $\chi: \neg \{\} \models \chi$  and  $\text{tot}\chi: \text{total-over-m } \{\} \ \{\chi\}$  and  $\chi\psi: \chi \in \text{fst } \psi$ 
    using  $H$  unfolding  $\text{tree}$  by  $\text{auto}$ 
    moreover have  $\{\#\} = \chi$ 
    using  $H$   $\text{atms-empty-iff-empty tot}\chi$ 
    unfolding  $\text{true-cls-def total-over-m-def total-over-set-def}$  by  $\text{fastforce}$ 
    moreover have  $\text{resolution}^{**} \ \psi \ \psi$  by  $\text{auto}$ 
    ultimately have  $?case$  by  $\text{metis}$ 
  }
moreover {
  fix  $v \ \text{tree1} \ \text{tree2}$ 
  assume  $\text{tree: tree} = \text{Node } v \ \text{tree1} \ \text{tree2}$ 
  obtain  $\psi_0$  where  $\psi_0: \text{resolution}^{**} \ \psi \ \psi_0$  and  $\text{simp: simplified } (\text{fst } \psi_0)$ 
  proof -
  { assume  $\text{simplified } (\text{fst } \psi)$ 
    moreover have  $\text{resolution}^{**} \ \psi \ \psi$  by  $\text{auto}$ 
    ultimately have  $\text{thesis}$  using  $\text{that}$  by  $\text{blast}$ 
  }
  moreover {
    assume  $\neg \text{simplified } (\text{fst } \psi)$ 
    then have  $\exists \psi'. \ \text{full1 simplify } (\text{fst } \psi) \ \psi'$ 
    by ( $\text{metis Nitpick.rtranclp-unfold bigger.prem}(3) \ \text{full1-def}$ 
       $\text{rtranclp-simplify-terminates}$ )
    then obtain  $N$  where  $\text{full1 simplify } (\text{fst } \psi) \ N$  by  $\text{metis}$ 
    then have  $\text{resolution } \psi \ (N, \text{snd } \psi)$ 
    using  $\text{resolution.intros}(1)[\text{of } \text{fst } \psi \ N \ \text{snd } \psi]$  by  $\text{auto}$ 
    moreover have  $\text{simplified } N$ 
    using  $\langle \text{full1 simplify } (\text{fst } \psi) \ N \rangle$  unfolding  $\text{full1-def}$  by  $\text{blast}$ 
    ultimately have  $?thesis$  using  $\text{that}$  by  $\text{force}$ 
  }
  ultimately show  $?thesis$  by  $\text{auto}$ 
}
qed

have  $p: \text{partial-interps tree } \{\} \ (\text{fst } \psi_0)$ 
and  $\text{uns: unsatisfiable } (\text{fst } \psi_0)$ 
and  $f: \text{finite } (\text{fst } \psi_0)$ 
and  $a\text{-}u\text{-}v: \text{already-used-inv } \psi_0$ 
  using  $\psi_0 \ \text{bigger.prem}(1) \ \text{rtranclp-resolution-preserve-partial-tree}$  apply  $\text{blast}$ 
  using  $\psi_0 \ \text{bigger.prem}(2) \ \text{rtranclp-resolution-preserves-unsat}$  apply  $\text{blast}$ 
  using  $\psi_0 \ \text{bigger.prem}(3) \ \text{rtranclp-resolution-finite}$  apply  $\text{blast}$ 
  using  $\text{rtranclp-resolution-already-used-inv}[OF \ \psi_0 \ \text{bigger.prem}(4)]$  by  $\text{blast}$ 
obtain  $\text{tree}' \ \psi'$  where
   $\text{inf: resolution}^{**} \ \psi_0 \ \psi'$  and
   $\text{part': partial-interps tree}' \ \{\} \ (\text{fst } \psi')$  and
   $\text{decrease: sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0$ 
  using  $\text{can-decrease-tree-size-resolution}[OF \ f \ a\text{-}u\text{-}v \ p \ \text{simp}]$  unfolding  $\text{tautology-def}$ 
  by  $\text{meson}$ 
have  $s: \text{sem-tree-size tree}' < \text{sem-tree-size tree}$  using  $\text{decrease}$  unfolding  $\text{tree}$  by  $\text{auto}$ 
have  $\text{fin: finite } (\text{fst } \psi')$ 
  using  $f \ \text{inf rtranclp-resolution-finite}$  by  $\text{blast}$ 

```

```

    have unsat: unsatisfiable (fst  $\psi'$ )
      using rtrancpl-resolution-preserves-unsat inf uns by metis
    have a-u-i': already-used-inv  $\psi'$ 
      using a-u-v inf rtrancpl-resolution-already-used-inv[of  $\psi_0$   $\psi'$ ] by auto
    have ?case
      using inf rtrancpl-trans[of resolution]  $H(1)[OF s part' unsat fin a-u-i'] \psi_0$  by blast
  }
  ultimately show ?case by (cases tree, auto)
qed
qed

```

```

lemma resolution-preserves-already-used-inv:
  assumes resolution  $S S'$ 
  and already-used-inv  $S$ 
  shows already-used-inv  $S'$ 
  using assms
  apply (induct rule: resolution.induct)
  apply (rule full1-simplify-already-used-inv; simp)
  apply (rule full-simplify-already-used-inv, simp)
  apply (rule inference-preserves-already-used-inv, simp)
  apply blast
done

```

```

lemma rtrancpl-resolution-preserves-already-used-inv:
  assumes resolution**  $S S'$ 
  and already-used-inv  $S$ 
  shows already-used-inv  $S'$ 
  using assms
  apply (induct rule: rtrancpl-induct)
  apply simp
  using resolution-preserves-already-used-inv by fast

```

```

lemma resolution-completeness:
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes unsat:  $\neg \text{satisfiable (fst } \psi)$ 
  and finite: finite (fst  $\psi$ )
  and snd  $\psi = \{\}$ 
  shows  $\exists \psi'. (\text{resolution** } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$ 
proof -
  have already-used-inv  $\psi$  unfolding assms by auto
  then show ?thesis using assms resolution-completeness-inv by blast
qed

```

```

lemma rtrancpl-preserves-sat:
  assumes simplify**  $S S'$ 
  and satisfiable  $S$ 
  shows satisfiable  $S'$ 
  using assms apply induction
  apply simp
  by (meson satisfiable-carac satisfiable-def simplify-preserves-un-sat-eq)

```

```

lemma resolution-preserves-sat:
  assumes resolution  $S S'$ 
  and satisfiable (fst  $S$ )
  shows satisfiable (fst  $S'$ )

```


using *assms* **apply** (*induction rule: resolution.induct*)
using *rtrancplp-preserves-sat* *trancplp-into-rtrancplp* **unfolding** *full1-def* **apply** *fastforce*
by (*metis fst-conv full-def inference-preserves-un-sat rtrancplp-preserves-sat*
satisfiable-carac' satisfiable-def)

lemma *rtrancplp-resolution-preserves-sat*:
assumes *resolution** S S'*
and *satisfiable (fst S)*
shows *satisfiable (fst S')*
using *assms* **apply** (*induction rule: rtrancplp-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 ψ)*
using *assms* **by** (*meson rtrancplp-resolution-preserves-sat satisfiable-def true-cls-empty*
true-clss-def)

lemma *resolution-soundness-and-completeness*:
fixes $\psi :: 'v :: \text{linorder state}$
assumes *finite: finite (fst ψ)*
and *snd: snd $\psi = \{\}$*
shows $(\exists \psi'. (\text{resolution** } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable (fst } \psi)$
using *assms* *resolution-completeness resolution-soundness* **by** *metis*

lemma *simplified-falsity*:
assumes *simp: simplified ψ*
and $\{\#\} \in \psi$
shows $\psi = \{\{\#\}\}$
proof (*rule ccontr*)
assume $H: \neg ?thesis$
then obtain χ **where** $\chi \in \psi$ **and** $\chi \neq \{\#\}$ **using** *assms(2)* **by** *blast*
then have $\{\#\} \subsetneq \chi$ **by** (*simp add: mset-less-empty-nonempty*)
then have *simplify ψ ($\psi - \{\chi\}$)*
using *simplify.subsumption[OF assms(2) $\langle\{\#\} \subsetneq \chi\rangle \langle\chi \in \psi\rangle$* **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 *rtrancplp-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  $\chi s: \text{resolution}^{**} \psi (\chi s, a-u-v)$  and  $F: \{\#\} \in \chi s$  by auto
  { assume simplified  $\chi s$ 
    then have  $?B$  using simplified-falsity[OF - F]  $\chi s$  by blast
  }
  moreover {
    assume  $\neg \text{simplified } \chi s$ 
    then obtain  $\chi s'$  where full1 simplify  $\chi s \chi s'$ 
      by (metis  $\chi s$  assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite)
    then have  $\{\#\} \in \chi s'$ 
      unfolding full1-def by (meson F rtranclp-simplify-falsity-in-preserved
        trancplp-into-rtranclp)
    then have  $?B$ 
      by (metis  $\chi s$  (full1 simplify  $\chi s \chi s'$ ) fst-conv full1-simp resolution-always-simplified
        rtranclp.rtrancl-into-rtrancl simplified-falsity)
  }
  ultimately show  $?B$  by blast
qed

lemma resolution-soundness-and-completeness':
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes
    finite: finite (fst  $\psi$ ) and
    snd: snd  $\psi = \{\}$ 
  shows  $(\exists a-u-v. (\text{resolution}^{**} \psi (\{\#\}), a-u-v)) \longleftrightarrow \text{unsatisfiable } (\text{fst } \psi)$ 
    using assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone
    by metis

end

theory Partial-Annotated-Clausal-Logic
imports Partial-Clausal-Logic

begin

```

13 Partial Clausal Logic

We here define marked literals (that will be used in both DPLL and CDCL) and the entailment corresponding to it.

13.1 Marked Literals

13.1.1 Definition

```

datatype ('v, 'vl, 'mark) marked-lit =
  is-marked: Marked (lit-of: 'v literal) (level-of: 'vl) |
  is-proped: Propagated (lit-of: 'v literal) (mark-of: 'mark)

```

lemma *marked-lit-list-induct*[*case-names nil marked proped*]:

assumes $P \square$ **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 (*rename-tac a xs, case-tac a*) *auto*

lemma *is-marked-ex-Marked*:

$\text{is-marked } L \implies \exists K lwl. L = \text{Marked } K lwl$
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 set* \Rightarrow *'a literal set* **where**
lits-of Ls = *lit-of ' Ls*

abbreviation *lits-of-l* :: (*'a, 'b, 'c*) *marked-lit list* \Rightarrow *'a literal set* **where**
lits-of-l Ls \equiv *lit-of ' (set Ls)*

lemma *lits-of-l-empty*[*simp*]:

$\text{lits-of } \{\} = \{\}$
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-l-cons*[*simp*]:

$\text{lits-of-l } (L \# Ls) = \text{insert } (\text{lit-of } L) (\text{lits-of-l } Ls)$
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-l-append*[*simp*]:

$\text{lits-of } (l \cup l') = \text{lits-of } l \cup \text{lits-of } l'$
unfolding *lits-of-def* **by** *auto*

lemma *finite-lits-of-def*[*simp*]:

$\text{finite } (\text{lits-of-l } L)$
unfolding *lits-of-def* **by** *auto*

abbreviation *unmark* **where**

$\text{unmark} \equiv (\lambda a. \{\#\text{lit-of } a\#\})$

abbreviation *unmark-s* **where**

$\text{unmark-s } M \equiv \text{unmark } ' M$

abbreviation *unmark-l* **where**

$\text{unmark-l } M \equiv \text{unmark-s } (\text{set } M)$

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of*[*simp*]:

$\text{atms-of-ms } (\text{unmark-l } M') = \text{atm-of } ' \text{lits-of-l } M'$
unfolding *atms-of-ms-def lits-of-def* **by** *auto*

lemma *lits-of-l-empty-is-empty*[*iff*]:

$\text{lits-of-l } M = \{\} \longleftrightarrow M = \square$
by (*induct M*) *auto*

13.1.2 Entailment

definition $\text{true-annot} :: ('a, 'l, 'm) \text{ marked-lits} \Rightarrow 'a \text{ clause} \Rightarrow \text{bool}$ (**infix** \models_a 49) **where**
 $I \models_a C \longleftrightarrow (\text{lits-of-l } I) \models C$

definition $\text{true-annots} :: ('a, 'l, 'm) \text{ marked-lits} \Rightarrow 'a \text{ clauses} \Rightarrow \text{bool}$ (**infix** \models_{as} 49) **where**
 $I \models_{as} CC \longleftrightarrow (\forall C \in CC. I \models_a C)$

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

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

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

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

lemma $\text{true-annots-single-true-annot[iff]}$:
 $I \models_{as} \{C\} \longleftrightarrow I \models_a C$
unfolding true-annots-def **by** auto

lemma $\text{true-annot-insert-l[simp]}$:
 $M \models_a A \Longrightarrow L \# M \models_a A$
unfolding true-annot-def **by** auto

lemma $\text{true-annots-insert-l [simp]}$:
 $M \models_{as} A \Longrightarrow L \# M \models_{as} A$
unfolding true-annots-def **by** auto

lemma $\text{true-annots-union[iff]}$:
 $M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$
unfolding true-annots-def **by** auto

lemma $\text{true-annots-insert[iff]}$:
 $M \models_{as} \text{insert } a \ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$
unfolding true-annots-def **by** auto

Link between \models_{as} and \models_s :

lemma $\text{true-annots-true-cl}$:
 $I \models_{as} CC \longleftrightarrow \text{lits-of-l } I \models_s CC$
unfolding $\text{true-annots-def Ball-def true-annot-def true-clss-def}$ **by** auto

lemma $\text{in-lit-of-true-annot}$:
 $a \in \text{lits-of-l } M \longleftrightarrow M \models_a \{\#a\#\}$
unfolding $\text{true-annot-def lits-of-def}$ **by** auto

lemma $\text{true-annot-lit-of-notin-skip}$:
 $L \# M \models_a A \Longrightarrow \text{lit-of } L \notin \# A \Longrightarrow M \models_a A$

unfolding *true-annot-def true-clss-def* **by** *auto*

lemma *true-clss-singleton-lit-of-implies-incl*:

$I \models_s \text{unmark-l } MLs \implies \text{lits-of-l } MLs \subseteq I$

unfolding *true-clss-def lits-of-def* **by** *auto*

lemma *true-annot-true-clss-clss*:

$MLs \models_a \psi \implies \text{set } (\text{map } (\lambda a. \{\# \text{lit-of } a \#\}) MLs) \models_p \psi$

unfolding *true-annot-def true-clss-clss-def true-clss-def*

by (*auto dest: true-clss-singleton-lit-of-implies-incl*)

lemma *true-annots-true-clss-clss*:

$MLs \models_{as} \psi \implies \text{set } (\text{map } (\lambda a. \{\# \text{lit-of } a \#\}) MLs) \models_{ps} \psi$

by (*auto*

dest: true-clss-singleton-lit-of-implies-incl

simp add: true-clss-def true-annots-def true-annot-def lits-of-def true-clss-def

true-clss-clss-def)

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

$\text{map } (\lambda M. \text{Marked } M \ a) \ M \models_{as} N \iff \text{set } M \models_s N$

proof –

have *: *lit-of* ‘ $(\lambda M. \text{Marked } M \ a)$ ’ *set* $M = \text{set } M$ **unfolding** *lits-of-def* **by** *force*

show ?thesis **by** (*simp add: true-annots-true-clss **)

qed

lemma *true-annot-singleton[iff]*: $M \models_a \{\# L \#\} \iff L \in \text{lits-of-l } M$

unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annots-true-clss-clss*:

$A \models_{as} \Psi \implies \text{unmark-l } A \models_{ps} \Psi$

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*:

$M @ M' \models_a D \iff M' @ M \models_a D$

unfolding *true-annot-def* **by** (*simp add: Un-commute*)

lemma *true-annots-commute*:

$M @ M' \models_{as} D \iff M' @ M \models_{as} D$

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* $\Rightarrow \text{bool}$

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

lemma *true-annot-iff-marked-or-true-lit*:
defined-lit $I\ L \longleftrightarrow ((\text{lits-of-l } I) \models_l L \vee (\text{lits-of-l } I) \models_l -L)$
unfolding *defined-lit-def* **by** (*auto simp add: lits-of-def rev-image-eqI dest!: literal-is-lit-of-marked*)

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

lemma *defined-lit-map*:
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]*:
defined-lit $I\ (-L) \longleftrightarrow \text{defined-lit } I\ L$
unfolding *defined-lit-def* **by** *auto*

lemma *Marked-Propagated-in-iff-in-lits-of-l*:
defined-lit $I\ L \longleftrightarrow (L \in \text{lits-of-l } I \vee -L \in \text{lits-of-l } I)$
unfolding *lits-of-def* *defined-lit-def*
by (*auto simp: rev-image-eqI*) (*rename-tac x, case-tac x, auto*) $+$

lemma *consistent-add-undefined-lit-consistent[simp]*:
assumes
consistent-interp (*lits-of-l* Ls) **and**
undefined-lit $Ls\ L$
shows *consistent-interp* (*insert* L (*lits-of-l* Ls))
using *assms* **unfolding** *consistent-interp-def* **by** (*auto simp: Marked-Propagated-in-iff-in-lits-of-l*)

lemma *decided-empty[simp]*:

\neg defined-lit [] L
unfolding defined-lit-def **by** simp

13.2 Backtracking

fun backtrack-split :: ('v, 'l, 'm) marked-lits
 \Rightarrow ('v, 'l, 'm) marked-lits \times ('v, 'l, 'm) marked-lits **where**
backtrack-split [] = ([], []) |
backtrack-split (Propagated L P # mlits) = apfst ((op #) (Propagated L P)) (backtrack-split mlits) |
backtrack-split (Marked L l # mlits) = ([], Marked L l # mlits)

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

lemma backtrack-split-snd-hd-marked:
 $\text{snd } (\text{backtrack-split } l) \neq [] \implies \text{is-marked } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$
by (induct l rule: marked-lit-list-induct) auto

lemma backtrack-split-list-eq[simp]:
 $\text{fst } (\text{backtrack-split } l) @ (\text{snd } (\text{backtrack-split } l)) = l$
by (induct l rule: marked-lit-list-induct) auto

lemma backtrack-snd-empty-not-marked:
 $\text{backtrack-split } M = (M'', []) \implies \forall l \in \text{set } M. \neg \text{is-marked } l$
by (metis append-Nil2 backtrack-split-fst-not-marked backtrack-split-list-eq snd-conv)

lemma backtrack-split-some-is-marked-then-snd-has-hd:
 $\exists l \in \text{set } M. \text{is-marked } l \implies \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$
by (metis backtrack-snd-empty-not-marked list.exhaust prod.collapse)

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma backtrack-split-takeWhile-dropWhile:
 $\text{backtrack-split } M = (\text{takeWhile } (\text{Not } o \text{ is-marked}) M, \text{dropWhile } (\text{Not } o \text{ is-marked}) M)$
proof (induct M)
case Nil **show** ?case **by** simp
next
case (Cons L M) **then show** ?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) marked-lits \times ('a, 'l, 'm) marked-lits) list **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  $M = [] \longleftrightarrow \text{False}$ 
  by (induct  $M$ , simp) (rename-tac  $a$   $xs$ , case-tac  $a$ , auto)

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

lemma get-all-marked-decomposition-decomp:
   $\text{hd } (\text{get-all-marked-decomposition } S) = (a, c) \implies S = c @ a$ 
proof (induct  $S$  arbitrary:  $a$   $c$ )
  case Nil
  then show ?case by simp
next
  case (Cons  $x$   $A$ )
  then show ?case by (cases  $x$ ; cases  $\text{hd } (\text{get-all-marked-decomposition } A)$ ) auto
qed

lemma get-all-marked-decomposition-backtrack-split:
   $\text{backtrack-split } S = (M, M') \longleftrightarrow \text{hd } (\text{get-all-marked-decomposition } S) = (M', M)$ 
proof (induction  $S$  arbitrary:  $M$   $M'$ )
  case Nil
  then show ?case by auto
next
  case (Cons  $a$   $S$ )
  then show ?case using backtrack-split-takeWhile-dropWhile by (cases  $a$ ) force+
qed

lemma get-all-marked-decomposition-nil-backtrack-split-snd-nil:
   $\text{get-all-marked-decomposition } S = [([], A)] \implies \text{snd } (\text{backtrack-split } S) = []$ 
  by (simp add: get-all-marked-decomposition-backtrack-split sndI)

lemma get-all-marked-decomposition-length-1-fst-empty-or-length-1:
  assumes  $\text{get-all-marked-decomposition } M = (a, b) \# []$ 
  shows  $a = [] \vee (\text{length } a = 1 \wedge \text{is-marked } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$ 
  using assms
proof (induct  $M$  arbitrary:  $a$   $b$  rule: marked-lit-list-induct)
  case nil then show ?case by simp
next
  case (marked  $L$  mark  $M$ )
  then show ?case by simp
next
  case (proped  $L$  mark  $M$ )
  then show ?case by (cases  $\text{get-all-marked-decomposition } M$ ) force+
qed

lemma get-all-marked-decomposition-fst-empty-or-hd-in-M:
  assumes  $\text{get-all-marked-decomposition } M = (a, b) \# l$ 
  shows  $a = [] \vee (\text{is-marked } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$ 
  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  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
  and  $L \in \text{set } b$ 
  shows  $\neg \text{is-marked } L$ 
  using assms apply (induct  $M$  arbitrary:  $a$   $b$  rule: marked-lit-list-induct, simp)
  by (rename-tac  $L' l$   $xs$   $a$   $b$ , case-tac get-all-marked-decomposition  $xs$ ; fastforce)+

lemma tl-get-all-marked-decomposition-skip-some:
  assumes  $x \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } M1))$ 
  shows  $x \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } (M0 @ M1)))$ 
  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  $(x, y) = \text{hd } (\text{get-all-marked-decomposition } M1)$ 
  shows  $(x, y) \in \text{set } (\text{get-all-marked-decomposition } (M0 @ \text{Marked } K i \# M1))$ 
  using assms
proof (induct  $M0$ )
  case Nil
  then show ?case by auto
next
  case (Cons  $L$   $M0$ )
  then have  $xy: (x, y) \in \text{set } (\text{get-all-marked-decomposition } (M0 @ \text{Marked } K i \# M1))$  by blast
  show ?case
    proof (cases  $L$ )
    case (Marked  $l$   $m$ )
    then show ?thesis using  $xy$  by auto
    next
    case (Propagated  $l$   $m$ )
    then show ?thesis
      using  $xy$  Cons.prems
      by (cases get-all-marked-decomposition  $(M0 @ \text{Marked } K i \# M1)$ )
        (auto dest!: get-all-marked-decomposition-decomp
          arg-cong[of get-all-marked-decomposition - - hd])
    qed
  qed

lemma get-all-marked-decomposition-snd-union:
   $\text{set } M = \bigcup (\text{set 'snd 'set } (\text{get-all-marked-decomposition } M)) \cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
  (is ? $M$   $M = ?U$   $M \cup ?Ls$   $M$ )
proof (induct  $M$  arbitrary:)
  case Nil
  then show ?case by simp
next
  case (Cons  $L$   $M$ )
  show ?case
    proof (cases  $L$ )
    case (Marked  $a$   $l$ ) note  $L = \text{this}$ 
    then have  $L \in ?Ls$   $(L \# M)$  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)
      then show ?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') ⇒
  ∃ b'. (a, b' @ b) ∈ set (get-all-marked-decomposition (M @ M'))

```

apply (induction M rule: marked-lit-list-induct)

apply (metis append-Nil)

apply auto[]

by (rename-tac L' m xs, case-tac get-all-marked-decomposition (xs @ M')) auto

lemma *get-all-marked-decomposition-remove-unmark-ssed-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-unmark-ssed-length* **by** fastforce

lemma *get-all-marked-decomposition-last-choice:*

assumes $tl (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)) \neq []$

and $\forall l \in \text{set } M'. \neg \text{is-marked } l$

and $hd (tl (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M))) = (M0', M0)$

shows $hd (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$

using *assms* **by** (induct M' rule: marked-lit-list-induct) auto

lemma *get-all-marked-decomposition-except-last-choice-equal:*

assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$

shows $tl (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$

$= tl (tl (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)))$

using *assms* **by** (induct M' rule: marked-lit-list-induct) auto

lemma *get-all-marked-decomposition-hd-hd:*

assumes $\text{get-all-marked-decomposition } Ls = (M, C) \# (M0, M0') \# l$

shows $tl M = M0' @ M0 \wedge \text{is-marked } (hd M)$

using *assms*

proof (induct Ls arbitrary: M C M0 M0' l)

case Nil

then show ?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 = Marked L level

have Ls = M0' @ M0

using g a **by** (force intro: get-all-marked-decomposition-decomp)

then have $tl M = M0' @ M0 \wedge \text{is-marked } (hd M)$ **using** g a **by** auto

}

moreover {

```

fix L P
assume a: a = Propagated L P
have tl M = M0' @ M0 ∧ is-marked (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) ∈ set (get-all-marked-decomposition M)
  shows ∃ c. M = c @ b @ a
  using assms apply (induct M rule: marked-lit-list-induct)
  apply simp
  by (rename-tac L' m xs, 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) ∈ set (get-all-marked-decomposition M)
  shows set b ⊆ set M and set a ⊆ set M
  using assms get-all-marked-decomposition-exists-prepend by fastforce+

```

```

lemma get-all-marked-decomposition-exists-prepend':
  assumes (a, b) ∈ set (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 (rename-tac L' m xs, 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) ∈ set (get-all-marked-decomposition M)
  shows set a ∪ set b ⊆ set M
  using assms by force

```

```

lemma Marked-cons-in-get-all-marked-decomposition-append-Marked-cons:
  ∃ M1 M2. (Marked K i # M1, M2) ∈ set (get-all-marked-decomposition (c @ Marked K i # c'))
  apply (induction c rule: marked-lit-list-induct)
  apply auto[2]
  apply (rename-tac L m xs,
    case-tac hd (get-all-marked-decomposition (xs @ Marked K i # c')))
  apply (case-tac get-all-marked-decomposition (xs @ Marked K i # c'))
  by auto

```

```

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

```

```

lemma all-decomposition-implies-empty[iff]:
  all-decomposition-implies N [] unfolding all-decomposition-implies-def by auto

```

```

lemma all-decomposition-implies-single[iff]:
  all-decomposition-implies N [(Ls, seen)]
    ⇔ unmark-l Ls ∪ N ⊨ps unmark-l seen

```

```

unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-append[iff]:
  all-decomposition-implies  $N$  ( $S @ S'$ )
   $\longleftrightarrow$  (all-decomposition-implies  $N$   $S \wedge$  all-decomposition-implies  $N$   $S'$ )
unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-cons-pair[iff]:
  all-decomposition-implies  $N$  ( $(Ls, \text{seen}) \# S'$ )
   $\longleftrightarrow$  (all-decomposition-implies  $N$   $[(Ls, \text{seen})] \wedge$  all-decomposition-implies  $N$   $S'$ )
unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-cons-single[iff]:
  all-decomposition-implies  $N$  ( $l \# S'$ )  $\longleftrightarrow$ 
  (unmark-l (fst  $l$ )  $\cup N \models_{ps}$  unmark-l (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  $N \cup \{\{\#lit\text{-of } L\# \mid L. \text{is-marked } L \wedge L \in \text{set } M\}\}$ 
   $\models_{ps}$   $(\lambda a. \{\#lit\text{-of } a\# \}) ' \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-marked-decomposition } M))$ 
using assms
proof (induct length (get-all-marked-decomposition M) arbitrary: M)
  case 0
  then show ?case by auto
next
  case (Suc n) note  $IH = \text{this}(1)$  and  $\text{length} = \text{this}(2)$  and  $\text{decomp} = \text{this}(3)$ 
  consider
    (le1)  $\text{length } (\text{get-all-marked-decomposition } M) \leq 1$ 
  | (gt1)  $\text{length } (\text{get-all-marked-decomposition } M) > 1$ 
  by arith
  then show ?case
  proof cases
    case le1
    then obtain  $a$   $b$  where  $g: \text{get-all-marked-decomposition } M = (a, b) \# []$ 
    by (cases get-all-marked-decomposition M) auto
    moreover {
      assume  $a = []$ 
      then have ?thesis using Suc.premis g by auto
    }
    moreover {
      assume  $l: \text{length } a = 1$  and  $m: \text{is-marked } (\text{hd } a)$  and  $hd: \text{hd } a \in \text{set } M$ 
      then have  $(\lambda a. \{\#lit\text{-of } a\# \}) (\text{hd } a) \in \{\{\#lit\text{-of } L\# \mid L. \text{is-marked } L \wedge L \in \text{set } M\}\}$  by auto
      then have  $H: \text{unmark-l } a \cup N \subseteq N \cup \{\{\#lit\text{-of } L\# \mid L. \text{is-marked } L \wedge L \in \text{set } M\}\}$ 
      using  $l$  by (cases a) auto
      have  $f1: (\lambda m. \{\#lit\text{-of } m\# \}) ' \text{set } a \cup N \models_{ps} (\lambda m. \{\#lit\text{-of } m\# \}) ' \text{set } b$ 
      using decomp unfolding all-decomposition-implies-def g by simp
      have ?thesis
      apply (rule true-cls-cls-subset) using  $f1$   $H$   $g$  by auto
    }
  ultimately show ?thesis
  using get-all-marked-decomposition-length-1-fst-empty-or-length-1 by blast
next
  case gt1

```

```

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- $M$ :  $set\ M' \subseteq set\ M$ 
  using  $length$  by (induct  $M$  rule: marked-lit-list-induct) (auto simp: subset-insertI2)
let  $?d = \bigcup (set\ 'snd\ 'set\ (get-all-marked-decomposition\ M'))$ 
let  $?unM = \{unmark\ L \mid L.\ is-marked\ L \wedge L \in set\ M\}$ 
let  $?unM' = \{unmark\ L \mid L.\ is-marked\ L \wedge L \in set\ M'\}$ 
{
  assume  $n = 0$ 
  then have get-all-marked-decomposition  $M' = []$  using  $length'$  by auto
  then have  $?thesis$  using Suc.premis unfolding all-decomposition-implies-def  $Ls0$  by auto
}
moreover {
  assume  $n: n > 0$ 
  then obtain  $Ls1$   $seen1$   $l$  where
     $Ls1$ : get-all-marked-decomposition  $M' = (Ls1, seen1) \# l$ 
    using  $length'$  by (induct  $M'$  rule: marked-lit-list-induct) auto

  have all-decomposition-implies  $N$  (get-all-marked-decomposition  $M'$ )
    using decomp unfolding  $Ls0$  by auto
  then have  $N: N \cup ?unM' \models_{ps} unmark-s\ ?d$ 
    using IH length' by auto
  have  $l: N \cup ?unM' \subseteq N \cup ?unM$ 
    using  $M'$ -in- $M$  by auto
  from true-clss-clss-subset[OF this N]
  have  $\Psi N: N \cup ?unM \models_{ps} unmark-s\ ?d$  by auto
  have is-marked (hd  $Ls0$ ) and  $LS: tl\ Ls0 = seen1\ @\ Ls1$ 
    using get-all-marked-decomposition-hd-hd[of M] unfolding  $Ls0\ Ls1$  by auto

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

  {
    assume  $Ls0 \neq []$ 
    then have hd  $Ls0 \in set\ M$ 
      using get-all-marked-decomposition-fst-empty-or-hd-in-M  $Ls0$  by blast
    then have  $N \cup ?unM \models_p unmark\ (hd\ Ls0)$ 
      using  $\langle is-marked\ (hd\ Ls0) \rangle$  by (metis (mono-tags, lifting) UnCI mem-Collect-eq true-clss-clss-in)
  } note  $hd-Ls0 = this$ 

  have  $l: unmark\ '(\ ?d \cup \{L \mid L.\ is-marked\ L \wedge L \in set\ M'\}) = unmark-s\ ?d \cup ?unM'$ 
    by auto
  have  $N \cup ?unM' \models_{ps} unmark\ '(\ ?d \cup \{L \mid L.\ is-marked\ L \wedge L \in set\ M'\})$ 
    unfolding  $l$  using  $N$  by (auto simp: all-in-true-clss-clss)
  then have  $t: N \cup ?unM' \models_{ps} unmark-l\ (tl\ Ls0)$ 
    using  $M'$  unfolding  $LS\ LSM$  by auto
  then have  $N \cup ?unM \models_{ps} unmark-l\ (tl\ Ls0)$ 
    using  $M'$ -in- $M$  true-clss-clss-subset[OF - t, of  $N \cup ?unM$ ] by auto
  then have  $N \cup ?unM \models_{ps} unmark-l\ Ls0$ 
    using hd-Ls0 by (cases  $Ls0$ ) auto

  moreover have  $unmark-l\ Ls0 \cup N \models_{ps} unmark-l\ seen0$ 

```

```

    using decomp unfolding Ls0 by simp
    moreover have  $\bigwedge M Ma. (M::'a \text{ literal multiset set}) \cup Ma \models_{ps} M$ 
      by (simp add: all-in-true-clss-clss)
    ultimately have  $\Psi: N \cup ?unM \models_{ps} unmark-l \text{ seen0}$ 
      by (meson true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r)

    moreover have  $unmark \text{ ' (set seen0 } \cup ?d) = unmark-l \text{ seen0 } \cup unmark-s \text{ ' ?d}$ 
      by auto
    ultimately have ?thesis using  $\Psi N$  unfolding Ls0 by simp
  }
  ultimately show ?thesis by auto
qed
qed

```

lemma *all-decomposition-implies-propagated-lits-are-implied:*
assumes *all-decomposition-implies N (get-all-marked-decomposition M)*
shows $N \cup \{\{\#lit-of L\# \mid L. is-marked L \wedge L \in set M\}\} \models_{ps} unmark-l M$
(is ?I \models_{ps} ?A)
proof –
 have $?I \models_{ps} unmark-s \{L \mid L. is-marked L \wedge L \in set M\}$
 by (*auto intro: all-in-true-clss-clss*)
 moreover have $?I \models_{ps} unmark \text{ ' } \bigcup (set \text{ ' snd ' set (get-all-marked-decomposition M))}$
 using *all-decomposition-implies-trail-is-implied assms by blast*
 ultimately have $N \cup \{unmark m \mid m. is-marked m \wedge m \in set M\}$
 $\models_{ps} unmark \text{ ' } \bigcup (set \text{ ' snd ' set (get-all-marked-decomposition M))}$
 $\cup unmark \text{ ' } \{m \mid m. is-marked m \wedge m \in set M\}$
 by *blast*
 then show *?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 $CNot :: 'v \text{ clause} \Rightarrow 'v \text{ clauses}$ **where**
 $CNot \psi = \{\{\#-L\# \mid L. L \in \# \psi\}\}$

lemma *in-CNot-uminus[iff]:*
shows $\{\#L\# \in CNot \psi \longleftrightarrow -L \in \# \psi$
unfolding *CNot-def* by *force*

lemma
shows
CNot-singleton[simp]: $CNot \{\#L\# \} = \{\{\#-L\#\}\}$ **and**
CNot-empty[simp]: $CNot \{\#\} = \{\}$ **and**
CNot-plus[simp]: $CNot (A + B) = CNot A \cup CNot B$
unfolding *CNot-def* by *auto*

lemma *CNot-eq-empty[iff]:*
 $CNot D = \{\} \longleftrightarrow D = \{\#\}$
unfolding *CNot-def* by (*auto simp add: multiset-eqI*)

lemma *in-CNot-implies-uminus:*

assumes $L \in \# D$ **and** $M \models_{as} CNot D$
shows $M \models_a \{\#-L\# \}$ **and** $-L \in lits-of-l M$
using *assms* **by** (*auto simp: true-annots-def true-annot-def CNot-def*)

lemma *CNot-remdups-mset[simp]:*
 $CNot (remdups-mset A) = CNot A$
unfolding *CNot-def* **by** *auto*

lemma *Ball-CNot-Ball-mset[simp] :*
 $(\forall x \in CNot D. P x) \longleftrightarrow (\forall L \in \# D. P \{\#-L\# \})$
unfolding *CNot-def* **by** *auto*

lemma *consistent-CNot-not:*
assumes *consistent-interp I*
shows $I \models_s CNot \varphi \implies \neg I \models \varphi$
using *assms* **unfolding** *consistent-interp-def true-clss-def true-cls-def* **by** *auto*

lemma *total-not-true-cls-true-clss-CNot:*
assumes *total-over-m I {φ}* **and** $\neg I \models \varphi$
shows $I \models_s CNot \varphi$
using *assms* **unfolding** *total-over-m-def total-over-set-def true-clss-def true-cls-def CNot-def*
apply *clarify*
by (*rename-tac x L, 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 $atm-of L \in atm-of \text{'lits-of-l } M$
by (*metis assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton*)

lemma *true-annots-CNot-all-uminus-atms-defined:*
assumes $M \models_{as} CNot T$ **and** $a1: -L \in \# T$
shows $atm-of L \in atm-of \text{'lits-of-l } M$
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  $I$  ( $B \cup \text{CNot } \{\#L\# \}$ ) and
  cons: consistent-interp  $I$  and
   $I: I \models_s B$ 
have total-over-m  $I$  ( $\{\{\#L\#\} \cup B \}$ ) using tot by auto
then have  $\neg I \models_s \text{insert } \{\#L\# \} B$ 
  using assms cons unfolding true-clss-clss-def by simp
then show  $I \models_s \text{CNot } \{\#L\# \}$ 
  using tot  $I$  by (cases  $L$ ) auto
qed

lemma true-annots-true-clss-def-iff-negation-in-model:
   $M \models_{as} \text{CNot } C \longleftrightarrow (\forall L \in \# C. \neg L \in \text{lits-of-l } M)$ 
  unfolding CNot-def true-annots-true-clss true-clss-def by auto

lemma consistent-CNot-not-tautology:
  consistent-interp  $M \implies M \models_s \text{CNot } D \implies \neg \text{tautology } D$ 
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 ( $\text{CNot } CC$ ) = atms-of-ms  $\{CC\}$ 
by simp

lemma total-over-m-CNot-total-over-m[simp]:
  total-over-m  $I$  ( $\text{CNot } C$ ) = total-over-set  $I$  (atms-of  $C$ )
  unfolding total-over-m-def total-over-set-def by auto

lemma uminus-lit-swap:  $\neg(a::'a \text{ literal}) = i \longleftrightarrow a = \neg i$ 
by auto

lemma true-clss-clss-plus-CNot:
  assumes  $CC\text{-}L: A \models_p CC + \{\#L\#\}$ 
  and  $\text{CNot}\text{-}CC: A \models_{ps} \text{CNot } CC$ 
  shows  $A \models_p \{\#L\#\}$ 
  unfolding true-clss-clss-def true-clss-clss-def CNot-def total-over-m-def
proof (intro allI impI)
  fix  $I$ 
  assume
    tot: total-over-set  $I$  (atms-of-ms ( $A \cup \{\{\#L\#\} \}$ )) and
    cons: consistent-interp  $I$  and
     $I: I \models_s A$ 
  let  $?I = I \cup \{Pos P \mid P. P \in \text{atms-of } CC \wedge P \notin \text{atm-of } 'I\}$ 
  have cons': consistent-interp  $?I$ 
    using cons unfolding consistent-interp-def
    by (auto simp: 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  $?I$  ( $A \cup \text{CNot } CC$ )
    using tot atms-of-s-def by (fastforce simp: total-over-m-def total-over-set-def)

  then have 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
  then have  $?I \models CC + \{\#L\#\}$  using  $CC\text{-}L$  cons'  $I'$  unfolding true-clss-clss-def by blast
  moreover
    have  $?I \models_s \text{CNot } CC$  using  $\text{CNot}\text{-}CC$  cons'  $I'$  tot-CNot unfolding true-clss-clss-def by auto

```


then have $\neg A \models_p CC$
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-clss-def)
then have $\neg ?I \models CC$ **using** $\langle ?I \models s \text{ CNot } CC \rangle$ *cons'* *consistent-CNot-not* **by** *blast*
ultimately have $?I \models \{\#L\# \}$ **by** *blast*
then show $I \models \{\#L\# \}$
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 $LM: L \# M \models_{as} \text{CNot } A$ **and** $LA: \text{lit-of } L \notin \# A \rightarrow \text{lit-of } L \notin \# A$
shows $M \models_{as} \text{CNot } A$
using LM **unfolding** *true-annots-def Ball-def*
proof (*intro allI impI*)
fix l
assume $H: \forall x. x \in \text{CNot } A \longrightarrow L \# M \models_a x$ **and** $l: l \in \text{CNot } A$
then have $L \# M \models_a l$ **by** *auto*
then show $M \models_a l$ **using** $LA \ l$ **by** (*cases L*) (*auto simp: CNot-def*)
qed

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:
 $A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} \text{CNot } B$
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** $\text{atm-of } (\text{lit-of } a) \notin \text{atms-of } D$
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** $\forall x \in \text{atms-of } D. x \notin \text{atm-of } \text{'lits-of-l } M$
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** $\forall x \in \text{atms-of-ms } D. x \notin \text{atm-of } \text{'lits-of-l } M$
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
 $\forall s \in \text{atms-of-ms } \{B\}. s \in \text{atm-of } \text{'lits-of-l } A$ **and**
 $\neg A \models_a B$
shows $A \models_{as} \text{CNot } B$
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** $\neg \text{lits-of-l } A \models_l \neg L$
then have $\text{atm-of } L \in \text{atm-of } \text{'lits-of-l } A$
using *assms(1)* **by** (*simp add: atm-of-lit-in-atms-of lits-of-def*)
then have $L \in \text{lits-of-l } A \vee \neg L \in \text{lits-of-l } A$

using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* by *metis*
 then have $L \in \text{ lits-of-l } A$ using $\langle \neg \neg \text{ lits-of-l } A \models l - L \rangle$ by *auto*
 then show *False*
 using *LB assms(2)* unfolding *true-annot-def true-lit-def true-cls-def Bex-def*
 by *blast*
 qed

lemma *CNot-union-mset[simp]*:
 $CNot (A \# \cup B) = CNot A \cup CNot B$
 unfolding *CNot-def* by *auto*

13.5 Other

abbreviation *no-dup* $L \equiv \text{distinct } (\text{map } (\lambda l. \text{ atm-of } (\text{lit-of } l)) L)$

lemma *no-dup-rev[simp]*:
 $\text{no-dup } (\text{rev } M) \longleftrightarrow \text{no-dup } M$
 by (*auto simp: rev-map[symmetric]*)

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
 assumes *no-dup* M
 shows $\text{length } M = \text{card } (\text{atm-of } ' \text{ lits-of-l } M)$
 using *assms* unfolding *lits-of-def* by (*induct M*) (*auto simp add: image-image*)

lemma *distinct-consistent-interp*:
 $\text{no-dup } M \implies \text{consistent-interp } (\text{ lits-of-l } M)$

proof (*induct M*)

case *Nil*
 show ?case by *auto*

next

case (*Cons L M*)
 then have $a1: \text{consistent-interp } (\text{ lits-of-l } M)$ 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 *Cons.prem*s by *auto*
 have *undefined-lit* M (*lit-of L*)
 using $a2$ *image-iff* unfolding *defined-lit-def* by *fastforce*
 then show ?case
 using $a1$ by *simp*

qed

lemma *distinct-get-all-marked-decomposition-no-dup*:
 assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
 and *no-dup* M
 shows *no-dup* $(a @ b)$
 using *assms* by *force*

lemma *true-annots-lit-of-notin-skip*:

assumes $L \# M \models_{as} CNot A$
 and $\neg \text{lit-of } L \notin \# A$
 and *no-dup* $(L \# M)$
 shows $M \models_{as} CNot A$

proof –

have $\forall l \in \# A. \neg l \in \text{ lits-of-l } (L \# M)$
 using *assms(1)* *in-CNot-implies-uminus(2)* by *blast*

moreover

have $\text{atm-of } (\text{lit-of } L) \notin \text{atm-of } ' \text{ lits-of-l } M$
 using *assms(3)* unfolding *lits-of-def* by *force*

then have $- \text{lit-of } L \notin \text{ lits-of-l } M$ **unfolding** *lits-of-def*
 by (*metis* (*no-types*) *atm-of-uminus imageI*)
 ultimately have $\forall l \in \# A. -l \in \text{ lits-of-l } M$
 using *assms(2)* by (*metis insertE lits-of-l-cons uminus-of-uminus-id*)
 then show *?thesis* by (*auto simp add: true-annots-def*)
 qed

type-synonym *'v clauses = 'v clause multiset*

abbreviation *true-annots-mset* (**infix** \models_{asm} 50) **where**
 $I \models_{asm} C \equiv I \models_{as} (\text{set-mset } C)$

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-mm :: 'a literal multiset multiset \Rightarrow 'a set* **where**
 $\text{atms-of-mm } U \equiv \text{atms-of-ms } (\text{set-mset } U)$

Other definition using *Union-mset*

lemma *atms-of-mm* $U \equiv \text{set-mset } (\bigcup \# \text{ image-mset } (\text{image-mset } \text{atm-of}) U)$
unfolding *atms-of-ms-def* by (*auto simp: atms-of-def*)

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-Abstract-Clause-Representation*

imports *Main Partial-Annotated-Clausal-Logic*

begin

13.6 Abstract Clause Representation

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

We assume the following:

- there is an equivalent to adding and removing a literal and to taking the union of clauses.

```

locale raw-cls =
  fixes
    mset-cls:: 'cls  $\Rightarrow$  'v clause and
    union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
    insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
    remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls
  assumes
    insert-cls[simp]: mset-cls (insert-cls L C) = mset-cls C + {#L#} and
    mset-cls-union-cls[simp]: mset-cls (union-cls C D) = mset-cls C # $\cup$  mset-cls D and
    remove-lit[simp]: mset-cls (remove-lit L C) = remove1-mset L (mset-cls C)
begin
end

```

This is a copy of an unnamed theorem of `~~/src/HOL/Library/Multiset.thy`.

```

lemma union-mset-list:
  mset xs # $\cup$  mset ys =
    mset (case-prod append (fold ( $\lambda x$  (ys, zs). (remove1 x ys, x # zs)) xs (ys, [])))
proof -
  have  $\bigwedge$ zs. mset (case-prod append (fold ( $\lambda x$  (ys, zs). (remove1 x ys, x # zs)) xs (ys, zs))) =
    (mset xs # $\cup$  mset ys) + mset zs
  by (induct xs arbitrary: ys) (simp-all add: multiset-eq-iff)
  then show ?thesis by simp
qed

```

Instantiation of the previous locale, in an unnamed context to avoid polluting with simp rules

```

context
begin
  interpretation list-cls: raw-cls mset
     $\lambda$ xs ys. case-prod append (fold ( $\lambda x$  (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
    op # remove1
    by unfold-locale (auto simp: union-mset-list ex-mset)

  interpretation cls-cls: raw-cls id
    op # $\cup$   $\lambda L$  C. C + {#L#} remove1-mset
    by unfold-locale (auto simp: union-mset-list)
end

```

Over the abstract clauses, we have the following properties:

- We can insert a clause
- We can take the union (used only in proofs for the definition of *clauses*)
- there is an operator indicating whether the abstract clause is contained or not
- if a concrete clause is contained the abstract clauses, then there is an abstract clause

```

locale raw-clss =
  raw-cls mset-cls union-cls insert-cls remove-lit
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls +
fixes

```

```

mset-clss:: 'clss  $\Rightarrow$  'v clauses and
union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss
assumes
insert-clss[simp]: mset-clss (insert-clss L C) = mset-clss C + {#mset-cls L#} and
union-clss[simp]: mset-clss (union-clss C D) = mset-clss C + mset-clss D and
mset-clss-union-clss[simp]: mset-clss (insert-clss C' D) = {#mset-cls C'#} + mset-clss D and
in-clss-mset-clss[dest]: in-clss a C  $\implies$  mset-cls a  $\in$  # mset-clss C and
in-mset-clss-exists-preimage: b  $\in$  # mset-clss C  $\implies$   $\exists$  b'. in-clss b' C  $\wedge$  mset-cls b' = b and
remove-from-clss-mset-clss[simp]:
  mset-clss (remove-from-clss a C) = mset-clss C - {#mset-cls a#} and
in-clss-union-clss[simp]:
  in-clss a (union-clss C D)  $\longleftrightarrow$  in-clss a C  $\vee$  in-clss a D
begin

end

experiment
begin
fun remove-first where
  remove-first - [] = [] |
  remove-first C (C' # L) = (if mset C = mset C' then L else C' # remove-first C L)

lemma remove1-mset-single-add:
  a  $\neq$  b  $\implies$  remove1-mset a ({#b#} + C) = {#b#} + remove1-mset a C
  remove1-mset a ({#a#} + C) = C
  by (auto simp: multiset-eq-iff)

lemma mset-map-mset-remove-first:
  mset (map mset (remove-first a C)) = remove1-mset (mset a) (mset (map mset C))
  by (induction C) (auto simp: ac-simps remove1-mset-single-add)

interpretation clss-clss: raw-clss id op # $\cup$   $\lambda$ L C. C + {#L#} remove1-mset
  id op + op  $\in$  #  $\lambda$ L C. C + {#L#} remove1-mset
  by unfold-locales (auto simp: ac-simps)

interpretation list-clss: raw-clss mset
   $\lambda$ xs ys. case-prod append (fold ( $\lambda$ x (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
  op # remove1  $\lambda$ L. mset (map mset L) op @  $\lambda$ L C. L  $\in$  set C op #
  remove-first
  by unfold-locales (auto simp: ac-simps union-mset-list mset-map-mset-remove-first ex-mset)
end

end
theory CDCL-WNOT-Measure
imports Main
begin

```

14 Measure

This measure show the termination of the core of CDCL: each step improves the number of literals we know for sure.

This measure can also be seen as the increasing lexicographic order: it is an order on bounded sequences, when each element is bounded. The proof involves a measure like the one defined here (the same?).

definition $\mu_C :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat}$ **where**
 $\mu_C \ s \ b \ M \equiv (\sum i=0..<\text{length } M. M!i * b^\wedge (s + i - \text{length } M))$

lemma $\mu_C\text{-nil}[simp]$:
 $\mu_C \ s \ b \ [] = 0$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma $\mu_C\text{-single}[simp]$:
 $\mu_C \ s \ b \ [L] = L * b^\wedge (s - \text{Suc } 0)$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma $\text{set-sum-atLeastLessThan-add}$:
 $(\sum i=k..<k+(b::\text{nat}). f \ i) = (\sum i=0..<b. f \ (k + i))$
by (*induction b*) *auto*

lemma $\text{set-sum-atLeastLessThan-Suc}$:
 $(\sum i=1..<\text{Suc } j. f \ i) = (\sum i=0..<j. f \ (\text{Suc } i))$
using $\text{set-sum-atLeastLessThan-add}[of \ - \ 1 \ j]$ **by** *force*

lemma $\mu_C\text{-cons}$:
 $\mu_C \ s \ b \ (L \# M) = L * b^\wedge (s - 1 - \text{length } M) + \mu_C \ s \ b \ M$
proof –
have $\mu_C \ s \ b \ (L \# M) = (\sum i=0..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$
unfolding $\mu_C\text{-def}$ **by** *blast*
also have $\dots = (\sum i=0..<1. (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$
 $+ (\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$
by (*rule setsum-add-nat-ivl[symmetric]*) *simp-all*
finally have $\mu_C \ s \ b \ (L \# M) = L * b^\wedge (s - 1 - \text{length } M)$
 $+ (\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$
by *auto*
moreover {
have $(\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M))) =$
 $(\sum i=0..<\text{length } (M). (L\#M)!(\text{Suc } i) * b^\wedge (s + (\text{Suc } i) - \text{length } (L\#M)))$
unfolding length-Cons $\text{set-sum-atLeastLessThan-Suc}$ **by** *blast*
also have $\dots = (\sum i=0..<\text{length } (M). M!i * b^\wedge (s + i - \text{length } M))$
by *auto*
finally have $(\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M))) = \mu_C \ s \ b \ M$
unfolding $\mu_C\text{-def}$.
}
ultimately show *?thesis* **by** *presburger*
qed

lemma $\mu_C\text{-append}$:
assumes $s \geq \text{length } (M @ M')$
shows $\mu_C \ s \ b \ (M @ M') = \mu_C \ (s - \text{length } M') \ b \ M + \mu_C \ s \ b \ M'$
proof –
have $\mu_C \ s \ b \ (M @ M') = (\sum i=0..<\text{length } (M @ M'). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')))$
unfolding $\mu_C\text{-def}$ **by** *blast*
moreover then have $\dots = (\sum i=0..<\text{length } M. (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')))$
 $+ (\sum i=\text{length } M..<\text{length } (M @ M'). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')))$
by (*auto intro!: setsum-add-nat-ivl[symmetric]*)
moreover

have $\forall i \in \{0..< \text{length } M\}. (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')) = M ! i * b^\wedge (s - \text{length } M' + i - \text{length } M)$
using $\langle s \geq \text{length } (M @ M') \rangle$ **by** $(\text{auto simp add: nth-append ac-simps})$
then have $\mu_C (s - \text{length } M') b M = (\sum i=0..< \text{length } M. (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')))$
unfolding $\mu_C\text{-def}$ **by** auto
ultimately have $\mu_C s b (M @ M') = \mu_C (s - \text{length } M') b M + (\sum i=\text{length } M..< \text{length } (M @ M'). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')))$
by auto
moreover {
have $(\sum i=\text{length } M..< \text{length } (M @ M'). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')) = (\sum i=0..< \text{length } M'. M ! i * b^\wedge (s + i - \text{length } M'))$
unfolding $\text{length-append set-sum-atLeastLessThan-add}$ **by** auto
then have $(\sum i=\text{length } M..< \text{length } (M @ M'). (M @ M')!i * b^\wedge (s + i - \text{length } (M @ M')) = \mu_C s b M'$
unfolding $\mu_C\text{-def}$.
}
ultimately show $?thesis$ **by** presburger
qed

lemma $\mu_C\text{-cons-non-empty-inf}$:
assumes $M\text{-ge-1}: \forall i \in \text{set } M. i \geq 1$ **and** $M: M \neq []$
shows $\mu_C s b M \geq b^\wedge (s - \text{length } M)$
using assms **by** $(\text{cases } M) (\text{auto simp: mult-eq-if } \mu_C\text{-cons})$

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 $(\text{cases } k = 0)$
apply $(\text{cases } n; \text{simp})$
by $(\text{induct } n) (\text{auto simp: Nat.nat-distrib})$

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

lemma $\mu_C\text{-bounded-non-degenerated}$:
fixes $b :: nat$
assumes
 $b > 0$ **and**
 $M \neq []$ **and**
 $M\text{-le}: \forall i < \text{length } M. M ! i < b$ **and**
 $s \geq \text{length } M$
shows $\mu_C s b M < b^\wedge s$
proof –
consider $(b1) \ b = 1 \mid (b) \ b > 1$ **using** $\langle b > 0 \rangle$ **by** $(\text{cases } b) \text{ auto}$
then show $?thesis$
proof cases
case $b1$
then have $\forall i < \text{length } M. M ! i = 0$ **using** $M\text{-le}$ **by** auto
then have $\mu_C s b M = 0$ **unfolding** $\mu_C\text{-def}$ **by** auto
then show $?thesis$ **using** $\langle b > 0 \rangle$ **by** auto
next
case b
have $\forall i \in \{0..< \text{length } M\}. M ! i * b^\wedge (s + i - \text{length } M) \leq (b-1) * b^\wedge (s + i - \text{length } M)$
using $M\text{-le } \langle b > 1 \rangle$ **by** auto
then have $\mu_C s b M \leq (\sum i=0..< \text{length } M. (b-1) * b^\wedge (s + i - \text{length } M))$
using $\langle M \neq [] \rangle \langle b > 0 \rangle$ **unfolding** $\mu_C\text{-def}$ **by** $(\text{auto intro: setsum-mono})$

```

also
  have  $\forall i \in \{0..<\text{length } M\}. (b-1) * b^{\wedge} (s+i - \text{length } M) = (b-1) * b^{\wedge} i * b^{\wedge} (s - \text{length } M)$ 
    by (metis Nat.add-diff-assoc2 add.commute assms(4) mult.assoc power-add)
  then have  $(\sum i=0..<\text{length } M. (b-1) * b^{\wedge} (s+i - \text{length } M))$ 
    =  $(\sum i=0..<\text{length } M. (b-1) * b^{\wedge} i * b^{\wedge} (s - \text{length } M))$ 
    by (auto simp add: ac-simps)
  also have  $\dots = (\sum i=0..<\text{length } M. b^{\wedge} i) * b^{\wedge} (s - \text{length } M) * (b-1)$ 
    by (simp add: setsum-left-distrib setsum-right-distrib ac-simps)
  finally have  $\mu_C s b M \leq (\sum i=0..<\text{length } M. b^{\wedge} i) * (b-1) * b^{\wedge} (s - \text{length } M)$ 
    by (simp add: ac-simps)

also
  have  $(\sum i=0..<\text{length } M. b^{\wedge} i) * (b-1) = b^{\wedge} (\text{length } M) - 1$ 
    using sum-of-powers[of b length M]  $\langle b > 1 \rangle$ 
    by (auto simp add: ac-simps)
  finally have  $\mu_C s b M \leq (b^{\wedge} (\text{length } M) - 1) * b^{\wedge} (s - \text{length } M)$ 
    by auto
  also have  $\dots < b^{\wedge} (\text{length } M) * b^{\wedge} (s - \text{length } M)$ 
    using  $\langle b > 1 \rangle$  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
qed

```

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

```

lemma  $\mu_C$ -bounded:
  fixes  $b :: \text{nat}$ 
  assumes
     $M\text{-le}: \forall i < \text{length } M. M!i < b$  and
     $s \geq \text{length } M$ 
     $b > 0$ 
  shows  $\mu_C s b M < b^{\wedge} s$ 
proof -
  consider ( $M0$ )  $M = [] \mid (M) b > 0$  and  $M \neq []$ 
  using  $M\text{-le}$  by (cases  $b$ , cases  $M$ ) auto
  then show ?thesis
  proof cases
    case  $M0$ 
    then show ?thesis using  $M\text{-le}$   $\langle b > 0 \rangle$  by auto
  next
    case  $M$ 
    show ?thesis using  $\mu_C$ -bounded-non-degenerated[OF M assms(1,2)] by arith
  qed
qed

```

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

```

lemma  $\mu_C$ -base-0:
  assumes  $\text{length } M \leq s$ 
  shows  $\mu_C s 0 M \leq M!0$ 
proof -
  {
    assume  $s = \text{length } M$ 
    moreover {

```



```

    fix n
    have (∑ i=0..<n. M ! i * (0::nat) ^ i) ≤ M ! 0
      apply (induction n rule: nat-induct)
      by simp (rename-tac n, case-tac n, auto)
  }
  ultimately have ?thesis unfolding μC-def by auto
}
moreover
{
  assume length M < s
  then have μC s 0 M = 0 unfolding μC-def by auto}
ultimately show ?thesis using assms unfolding μC-def by linarith
qed

end
theory CDCL-NOT
imports CDCL-Abstract-Clause-Representation List-More Wellfounded-More CDCL-WNOT-Measure
begin

```

15 NOT's CDCL

15.1 Auxiliary Lemmas and Measure

lemma *no-dup-cannot-not-lit-and-uminus*:
 $\text{no-dup } M \implies \neg \text{lit-of } xa = \text{lit-of } x \implies x \in \text{set } M \implies xa \notin \text{set } M$
 by (metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id')

lemma *true-clss-single-iff-incl*:
 $I \models_s \text{single } B \iff B \subseteq I$
 unfolding true-clss-def by auto

lemma *atms-of-ms-single-atm-of[simp]*:
 $\text{atms-of-ms } \{\text{unmark } L \mid L. P L\} = \text{atm-of } \{\text{lit-of } L \mid L. P L\}$
 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*:
 $\text{atms-of-ms } (\text{unmark } A) = \text{atm-of } (\text{lit-of } A)$
 unfolding atms-of-ms-def by auto

15.2 Initial definitions

15.2.1 The state

We define here an abstraction over operation on the state we are manipulating.

```

locale dpll-state-ops =
  raw-clss mset-clss union-clss insert-clss remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
for
  mset-clss:: 'cls ⇒ 'v clause and
  union-clss:: 'cls ⇒ 'cls ⇒ 'cls and
  insert-clss:: 'v literal ⇒ 'cls ⇒ 'cls and

```

```

remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
mset-clss :: 'clss  $\Rightarrow$  'v clauses and
union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss +
fixes
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st
begin

notation insert-cls (infix !++ 50)

notation in-clss (infix ! $\in$ ! 50)
notation union-clss (infix  $\oplus$  50)
notation insert-clss (infix !++! 50)

abbreviation clausesNOT where
clausesNOT S  $\equiv$  mset-clss (raw-clauses S)

end

locale dpll-state =
  dpll-state-ops mset-cls union-cls insert-cls remove-lit — related to each clause
  mset-clss union-clss in-clss insert-clss remove-from-clss — related to the clauses

  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT — related to the state
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss :: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st +
assumes
  trail-prepend-trail[simp]:
     $\bigwedge st L. \text{undefined-lit } (\text{trail } st) (\text{lit-of } L) \implies \text{trail } (\text{prepend-trail } L \text{ } st) = L \# \text{trail } st$ 
and
  tl-trail[simp]: trail (tl-trail S) = tl (trail S) and
  trail-add-clNOT[simp]:  $\bigwedge st C. \text{no-dup } (\text{trail } st) \implies \text{trail } (\text{add-cl}_{NOT} C \text{ } st) = \text{trail } st$  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]:
   $\bigwedge st L. \text{undefined-lit } (trail\ st) \ (lit\text{-of } L) \implies$ 
     $clauses_{NOT} \ (prepend\text{-trail } L\ st) = clauses_{NOT} \ st$ 
and
clauses-tl-trail[simp]:  $\bigwedge st. clauses_{NOT} \ (tl\text{-trail } st) = clauses_{NOT} \ st$  and
clauses-add-clsNOT[simp]:
   $\bigwedge st C. no\text{-dup } (trail\ st) \implies clauses_{NOT} \ (add\text{-cls}_{NOT} \ C\ st) = \{\#mset\text{-cls } C\# \} + clauses_{NOT} \ st$ 
and
clauses-remove-clsNOT[simp]:
   $\bigwedge st C. clauses_{NOT} \ (remove\text{-cls}_{NOT} \ C\ st) = removeAll\text{-mset } (mset\text{-cls } C) \ (clauses_{NOT} \ st)$ 
begin

function reduce-trail-toNOT :: 'a list  $\Rightarrow$  'st  $\Rightarrow$  'st where
reduce-trail-toNOT F S =
  (if length (trail S) = length F  $\vee$  trail S = [] then S else reduce-trail-toNOT F (tl-trail S))
by fast+
termination by (relation measure ( $\lambda(-, S). length \ (trail \ S)$ )) auto
declare reduce-trail-toNOT.simps[simp del]

lemma
shows
reduce-trail-toNOT-nil[simp]: trail S = []  $\implies$  reduce-trail-toNOT F S = S and
reduce-trail-toNOT-eq-length[simp]: length (trail S) = length F  $\implies$  reduce-trail-toNOT F S = S
by (auto simp: reduce-trail-toNOT.simps)

lemma reduce-trail-toNOT-length-ne[simp]:
  length (trail S)  $\neq$  length F  $\implies$  trail S  $\neq$  []  $\implies$ 
    reduce-trail-toNOT F S = reduce-trail-toNOT F (tl-trail S)
by (auto simp: reduce-trail-toNOT.simps)

lemma trail-reduce-trail-toNOT-length-le:
assumes length F > length (trail S)
shows trail (reduce-trail-toNOT F S) = []
using assms by (induction F S rule: reduce-trail-toNOT.induct)
(simp add: less-imp-diff-less reduce-trail-toNOT.simps)

lemma trail-reduce-trail-toNOT-nil[simp]:
  trail (reduce-trail-toNOT [] S) = []
by (induction [] S rule: reduce-trail-toNOT.induct)
(simp add: less-imp-diff-less reduce-trail-toNOT.simps)

lemma clauses-reduce-trail-toNOT-nil:
  clausesNOT (reduce-trail-toNOT [] S) = clausesNOT S
by (induction [] S rule: reduce-trail-toNOT.induct)
(simp add: less-imp-diff-less reduce-trail-toNOT.simps)

lemma trail-reduce-trail-toNOT-drop:
  trail (reduce-trail-toNOT F S) =
    (if length (trail S)  $\geq$  length F
     then drop (length (trail S) - length F) (trail S)
     else [])
apply (induction F S rule: reduce-trail-toNOT.induct)
apply (rename-tac F S, case-tac trail S)
apply auto[]
apply (rename-tac list, case-tac Suc (length list) > 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 $\text{trail } (\text{reduce-trail-to}_{\text{NOT}} F S) = F$
using *assms* **by** (*auto simp: trail-reduce-trail-to_{NOT}-drop*)

lemma *reduce-trail-to_{NOT}-clauses[*simp*]*:
 $\text{clauses}_{\text{NOT}} (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{clauses}_{\text{NOT}} S$
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**
 $\text{trail-weight } S \equiv \text{map } ((\lambda l. 1 + \text{length } l) \circ \text{snd}) (\text{get-all-marked-decomposition } (\text{trail } S))$

definition *state-eq_{NOT}* :: $'st \Rightarrow 'st \Rightarrow \text{bool}$ (*infix* \sim 50) **where**
 $S \sim T \longleftrightarrow \text{trail } S = \text{trail } T \wedge \text{clauses}_{\text{NOT}} S = \text{clauses}_{\text{NOT}} T$

lemma *state-eq_{NOT}-ref[*simp*]*:
 $S \sim S$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma *state-eq_{NOT}-sym*:
 $S \sim T \longleftrightarrow T \sim S$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma *state-eq_{NOT}-trans*:
 $S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma
shows
 $\text{state-eq}_{\text{NOT-trail}}: S \sim T \Longrightarrow \text{trail } S = \text{trail } T$ **and**
 $\text{state-eq}_{\text{NOT-clauses}}: S \sim T \Longrightarrow \text{clauses}_{\text{NOT}} S = \text{clauses}_{\text{NOT}} T$
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*:
 $\text{trail } S = \text{trail } T \Longrightarrow \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F T)$
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 $\text{clauses}_{\text{NOT}} (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{clauses}_{\text{NOT}} (\text{reduce-trail-to}_{\text{NOT}} F T)$
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]:
 no-dup (trail S) \implies
 trail (reduce-trail-to_{NOT} F (add-cl_{NOT} C S)) = trail (reduce-trail-to_{NOT} F S)
by (rule trail-eq-reduce-trail-to_{NOT}-eq) simp

lemma reduce-trail-to_{NOT}-trail-tl-trail-decomp[simp]:
 trail S = F' @ Marked K () # F \implies
 trail (reduce-trail-to_{NOT} F (tl-trail S)) = F
apply (rule reduce-trail-to_{NOT}-skip-beginning[of - tl (F' @ Marked K () # [])])
by (cases F') (auto simp add:tl-append reduce-trail-to_{NOT}-skip-beginning)

end

15.2.2 Definition of the operation

locale propagate-ops =
 dpll-state mset-cl_s union-cl_s insert-cl_s remove-lit
 mset-cl_{ss} union-cl_{ss} in-cl_{ss} insert-cl_{ss} remove-from-cl_{ss}
 trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
for
 mset-cl_s:: 'cl_s \Rightarrow 'v clause **and**
 union-cl_s:: 'cl_s \Rightarrow 'cl_s \Rightarrow 'cl_s **and**
 insert-cl_s:: 'v literal \Rightarrow 'cl_s \Rightarrow 'cl_s **and**
 remove-lit:: 'v literal \Rightarrow 'cl_s \Rightarrow 'cl_s **and**
 mset-cl_{ss}:: 'cl_{ss} \Rightarrow 'v clauses **and**
 union-cl_{ss}:: 'cl_{ss} \Rightarrow 'cl_{ss} \Rightarrow 'cl_{ss} **and**
 in-cl_{ss}:: 'cl_s \Rightarrow 'cl_{ss} \Rightarrow bool **and**
 insert-cl_{ss}:: 'cl_s \Rightarrow 'cl_{ss} \Rightarrow 'cl_{ss} **and**
 remove-from-cl_{ss}:: 'cl_s \Rightarrow 'cl_{ss} \Rightarrow 'cl_{ss} **and**
 trail:: 'st \Rightarrow ('v, unit, unit) marked-lits **and**
 raw-clauses:: 'st \Rightarrow 'cl_{ss} **and**
 prepend-trail:: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
 tl-trail:: 'st \Rightarrow 'st **and**
 add-cl_{NOT}:: 'cl_s \Rightarrow 'st \Rightarrow 'st **and**
 remove-cl_{NOT}:: 'cl_s \Rightarrow 'st \Rightarrow 'st +
fixes
 propagate-cond:: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool
begin
inductive propagate_{NOT}:: 'st \Rightarrow 'st \Rightarrow bool **where**
 propagate_{NOT}[intro]: C + {#L#} \in # clauses_{NOT} S \implies trail S \models as CNot C
 \implies undefined-lit (trail S) L
 \implies propagate-cond (Propagated L ()) S
 \implies T \sim prepend-trail (Propagated L ()) S
 \implies propagate_{NOT} S T
inductive-cases propagate_{NOT}E[elim]: propagate_{NOT} S T
end

locale decide-ops =
 dpll-state mset-cl_s union-cl_s insert-cl_s remove-lit
 mset-cl_{ss} union-cl_{ss} in-cl_{ss} insert-cl_{ss} remove-from-cl_{ss}
 trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
for

```

mset-cls:: 'cls  $\Rightarrow$  'v clause and
union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
mset-clss:: 'clss  $\Rightarrow$  'v clauses and
union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st

begin
inductive decideNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
decideNOT[intro]: undefined-lit (trail S) L  $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)
 $\Rightarrow$  T  $\sim$  prepend-trail (Marked L ()) S
 $\Rightarrow$  decideNOT S T

inductive-cases decideNOTE[elim]: decideNOT S S'
end

locale backjumping-ops =
  dpll-state mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'cls  $\Rightarrow$  '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
 $\Rightarrow$  T  $\sim$  prepend-trail (Propagated L ()) (reduce-trail-toNOT F S)
 $\Rightarrow$  C  $\in$  clausesNOT S
 $\Rightarrow$  trail S  $\models$ as CNot C
 $\Rightarrow$  undefined-lit F L
 $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))

```

$\Rightarrow \text{clauses}_{NOT} S \models_{pm} C' + \{\#L\# \}$
 $\Rightarrow F \models_{as} CNot C'$
 $\Rightarrow \text{backjump-conds } C C' L S T$
 $\Rightarrow \text{backjump } S T$

inductive-cases *backjumpE*: *backjump* *S T*

The condition *atm-of* *L* \in *atms-of-mm* (*clauses*_{NOT} *S*) \cup *atm-of* ‘*lits-of-l* (*trail S*) is not implied by the condition *clauses*_{NOT} *S* $\models_{pm} C' + \{\#L\# \}$ (no negation).

end

15.3 DPLL with backjumping

locale *dpll-with-backjumping-ops* =

propagate-ops *mset-cls* *union-cls* *insert-cls* *remove-lit*
mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*
trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls*_{NOT} *remove-cls*_{NOT} *propagate-conds* +
decide-ops *mset-cls* *union-cls* *insert-cls* *remove-lit*
mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*
trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls*_{NOT} *remove-cls*_{NOT} +
backjumping-ops *mset-cls* *union-cls* *insert-cls* *remove-lit*
mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*
trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls*_{NOT} *remove-cls*_{NOT} *backjump-conds*

for

mset-cls :: 'cls \Rightarrow 'v clause **and**
union-cls :: 'cls \Rightarrow 'cls \Rightarrow 'cls **and**
insert-cls :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**
remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**
mset-clss :: 'clss \Rightarrow 'v clauses **and**
union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss **and**
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool **and**
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**
raw-clauses :: 'st \Rightarrow 'clss **and**
prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
*add-cls*_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
*remove-cls*_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
inv :: 'st \Rightarrow bool **and**
backjump-conds :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool **and**
propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool +

assumes

bj-can-jump:
 $\bigwedge S C F' K F L.$
inv *S* \Rightarrow
no-dup (*trail S*) \Rightarrow
trail S = *F' @ Marked K () # F* \Rightarrow
 $C \in \# \text{clauses}_{NOT} S \Rightarrow$
trail S $\models_{as} CNot C \Rightarrow$
undefined-lit F L \Rightarrow
 $\text{atm-of } L \in \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of ' (lits-of-l (F' @ Marked K () # F))} \Rightarrow$
 $\text{clauses}_{NOT} S \models_{pm} C' + \{\#L\# \} \Rightarrow$
 $F \models_{as} CNot C' \Rightarrow$
 $\neg \text{no-step backjump } S$

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 $atm\text{-}of\ L \in atm\text{-}of\ ' lits\text{-}of\text{-}l\ (F' @ Marked\ K\ () \# F)$ is important, otherwise you are not sure that you can backtrack.

15.3.1 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 decideNOT propagateNOT backjump]`:

fixes `S T` :: `'st`

assumes

`dpll-bj S T and`

`inv S`

$\bigwedge L\ T.\ \text{undefined-lit}\ (\text{trail}\ S)\ L \Rightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (\text{clauses}_{NOT}\ S)$

$\Rightarrow T \sim \text{prepend-trail}\ (\text{Marked}\ L\ ())\ S$

$\Rightarrow P\ S\ T$ **and**

$\bigwedge C\ L\ T.\ C + \{\#L\# \} \in \# \text{ clauses}_{NOT}\ S \Rightarrow \text{trail}\ S \models_{as}\ CNot\ C \Rightarrow \text{undefined-lit}\ (\text{trail}\ S)\ L$

$\Rightarrow T \sim \text{prepend-trail}\ (\text{Propagated}\ L\ ())\ S$

$\Rightarrow P\ S\ T$ **and**

$\bigwedge C\ F'\ K\ F\ L\ C'\ T.\ C \in \# \text{ clauses}_{NOT}\ S \Rightarrow F' @ \text{Marked}\ K\ () \# F \models_{as}\ CNot\ C$

$\Rightarrow \text{trail}\ S = F' @ \text{Marked}\ K\ () \# F$

$\Rightarrow \text{undefined-lit}\ F\ L$

$\Rightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (\text{clauses}_{NOT}\ S) \cup atm\text{-}of\ ' (\text{lits-of-l}\ (F' @ \text{Marked}\ K\ () \# F))$

$\Rightarrow \text{clauses}_{NOT}\ S \models_{pm}\ C' + \{\#L\# \}$

$\Rightarrow F \models_{as}\ CNot\ C'$

$\Rightarrow T \sim \text{prepend-trail}\ (\text{Propagated}\ L\ ())\ (\text{reduce-trail-to}_{NOT}\ F\ S)$

$\Rightarrow P\ S\ T$

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 propagateNOTE`) **using** `assms(4)` **apply** `blast`

apply (`elim backjumpE`) **using** `assms(5)` `(inv S)` **by** `simp`

15.3.2 Basic properties

First, some better suited induction principle **lemma** `dpll-bj-clauses`:

assumes `dpll-bj S T and inv S`

shows `clausesNOT S = clausesNOT 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-toNOT-skip-beginning`)

Valuations lemma *dpll-bj-sat-iff*:
 assumes *dpll-bj S T* and *inv S*
 shows $I \models_{sm} clauses_{NOT} S \longleftrightarrow I \models_{sm} clauses_{NOT} T$
 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 $atms-of-mm (clauses_{NOT} S) = atms-of-mm (clauses_{NOT} T)$
 using *assms* by (induction rule: *dpll-bj-all-induct*) auto

lemma *dpll-bj-atms-in-trail*:
 assumes
 dpll-bj S T and
 inv S and
 $atm-of \text{ ' (lits-of-l (trail S)) } \subseteq atms-of-mm (clauses_{NOT} S)$
 shows $atm-of \text{ ' (lits-of-l (trail T)) } \subseteq atms-of-mm (clauses_{NOT} S)$
 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
 $atms-of-mm (clauses_{NOT} S) \subseteq A$ and
 $atm-of \text{ ' (lits-of-l (trail S)) } \subseteq A$
 shows $atm-of \text{ ' (lits-of-l (trail T)) } \subseteq A$
 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
 $decomp: all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))$
 shows $all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))$
 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* = $clauses_{NOT} S$

 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: unmark-l a  $\cup$  set-mset ?N  $\models_{ps}$  unmark-l (tl y)
  using decomp ay unfolding all-decomposition-implies-def by (simp add: M)+

moreover have unmark-l a  $\cup$  set-mset ?N  $\models_p$  {#L#} (is ?I  $\models_p$  -)
  proof (rule true-clss-clss-plus-CNot)
    show ?I  $\models_p$  C + {#L#}
      using propa propagateNOT.prems by (auto dest!: true-clss-clss-in-imp-true-clss-clss)
  next
    have ( $\lambda m. \{ \#lit\text{-of } m \# \}$ ) ‘ set ?M’  $\models_{ps}$  CNot C
      using (trail S  $\models_{as}$  CNot C) undef by (auto simp add: true-annots-true-clss-clss)
    have a1: ( $\lambda m. \{ \#lit\text{-of } m \# \}$ ) ‘ set a  $\cup$  ( $\lambda m. \{ \#lit\text{-of } m \# \}$ ) ‘ set (tl y)  $\models_{ps}$  CNot C
      using propagateNOT.hypos(2) tr-S true-annots-true-clss-clss
      by (force simp add: image-Un sup-commute)
    have a2: set-mset (clausesNOT S)  $\cup$  unmark-l a
       $\models_{ps}$  unmark-l (tl y)
      using calculation by (auto simp add: sup-commute)
    show ( $\lambda m. \{ \#lit\text{-of } m \# \}$ ) ‘ set a  $\cup$  set-mset (clausesNOT S)  $\models_{ps}$  CNot C
      proof -
        have set-mset (clausesNOT 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 (clausesNOT 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 unmark-l a  $\cup$  set-mset ?N  $\models_{ps}$  unmark-l ?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 (clausesNOT 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 unmark-l (fst (hd (get-all-marked-decomposition F)))
   $\cup$  set-mset (clausesNOT S)
   $\models_{ps}$  unmark-l (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-l F
    using (F  $\models_{as}$  CNot D) unfolding atms-of-def
    by (meson image-subsetI 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: unmark-l a  $\cup$  set-mset (clausesNOT S)  $\models_{ps}$  unmark-l b

```

```

using decomp unfolding all-decomposition-implies-def by (auto simp add: F)

have F-D: unmark-l F  $\models_{ps}$  CNot D
  using  $\langle F \models_{as} CNot D \rangle$  by (simp add: true-annots-true-clss-clss)
then have unmark-l a  $\cup$  unmark-l b  $\models_{ps}$  CNot D
  unfolding  $\langle F = b @ a \rangle$  by (simp add: image-Un sup commute)
have a-N-CNot-D: unmark-l a  $\cup$  set-mset (clausesNOT S)
 $\models_{ps}$  CNot D  $\cup$  unmark-l b
  apply (rule true-clss-clss-left-right)
  using a-N-b F-D unfolding  $\langle F = b @ a \rangle$  by (auto simp add: image-Un ac-simps)

have a-N-D-L: unmark-l a  $\cup$  set-mset (clausesNOT S)  $\models_p$  D + {#L#}
  by (simp add: N-C)
have unmark-l a  $\cup$  set-mset (clausesNOT 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

```

15.3.3 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:*
 $length (get_all_marked_decomposition M) \leq 1 + length M$
by (*induction M rule: marked-lit-list-induct*) *auto*

lemma *length-in-get-all-marked-decomposition-bounded:*

assumes $i:i \in \text{set } (\text{trail-weight } S)$
shows $i \leq \text{Suc } (\text{length } (\text{trail } S))$
proof –
obtain $a \ b$ **where**
 $(a, b) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ **and**
 $ib: i = \text{Suc } (\text{length } b)$
using i **by** *auto*
then obtain c **where** $\text{trail } S = c @ b @ a$
using *get-all-marked-decomposition-exists-prepend'* **by** *metis*
from *arg-cong[OF this, of length]* **show** *?thesis* **using** $i \ ib$ **by** *auto*
qed

Well-foundedness The bounds are the following:

- $1 + \text{card } (\text{atms-of-ms } A)$: $\text{card } (\text{atms-of-ms } A)$ 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 $\text{unassigned-lit} :: 'b \text{ literal multiset set} \Rightarrow 'a \text{ list} \Rightarrow \text{nat}$ **where**
 $\text{unassigned-lit } N \ M \equiv \text{card } (\text{atms-of-ms } N) - \text{length } M$

lemma *dpll-bj-trail-mes-increasing-prop*:

fixes $M :: ('v, \text{unit}, \text{unit}) \text{ marked-lits}$ **and** $N :: 'v \text{ clauses}$

assumes

$\text{dpll-bj } S \ T$ **and**

$\text{inv } S$ **and**

$\text{NA: atms-of-mm } (\text{clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A$ **and**

$\text{MA: atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

$\text{n-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } A$

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 ($\text{propagate}_{\text{NOT}} C \ L$) **note** $\text{CLN} = \text{this}(1)$ **and** $\text{MC} = \text{this}(2)$ **and** $\text{undef-L} = \text{this}(3)$ **and** $T = \text{this}(4)$

have $\text{incl: atm-of ' lits-of-l } (\text{Propagated } L \ ()) \# \text{trail } S \subseteq \text{atms-of-ms } A$

using $\text{propagate}_{\text{NOT}}.\text{hyps}$ $\text{propagate-ops.propagate}_{\text{NOT}}$ $\text{dpll-bj-atms-in-trail-in-set}$ $\text{bj-propagate}_{\text{NOT}}$

NA MA CLN **by** (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

have $\text{no-dup: no-dup } (\text{Propagated } L \ ()) \# \text{trail } S$

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 (*cases get-all-marked-decomposition (trail S) auto*)

have $\text{b-le-M: length } b \leq \text{length } (\text{trail } S)$

using $\text{get-all-marked-decomposition-decomp}$ [*of trail S*] **by** (*simp add: M*)

have $\text{finite } (\text{atms-of-ms } A)$ **using** finite **by** *simp*

then have $\text{length } (\text{Propagated } L \ ()) \# \text{trail } S \leq \text{card } (\text{atms-of-ms } A)$

using $\text{incl finite unfolding no-dup-length-eq-card-atm-of-lits-of-l}$ [*OF no-dup*]

by (*simp add: card-mono*)

```

then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L d # b))
  using b-le-M by auto
then show ?case using T undef-L by (auto simp: latm M  $\mu_C$ -cons)
next
case (decideNOT L) note undef-L = this(1) and MC = this(2) and T = this(3)
have incl: atm-of ' lits-of-l (Marked L () # (trail S))  $\subseteq$  atms-of-ms A
  using dpll-bj-atms-in-trail-in-set bj-decideNOT decideNOT.decideNOT[OF decideNOT.hyps] NA MA
MC
  by auto

have no-dup: no-dup (Marked L () # (trail S))
  using defined-lit-map n-d undef-L by auto
obtain a b l where M: get-all-marked-decomposition (trail S) = (a, b) # l
  by (cases get-all-marked-decomposition (trail S)) auto

then have length (Marked L () # (trail S))  $\leq$  card (atms-of-ms A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of-l[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-l (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-l[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-l)
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 (λa. Suc (length (snd a))) ((get-all-marked-decomposition F))
by (simp add: rev-map[symmetric] rev-swap)
have length (rev rem @ map (λa. Suc (length (snd a))) (get-all-marked-decomposition F))
  ≤ Suc (card (atms-of-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 ⇒ length xs - Suc i < length xs
  by auto
  then have H: i < length xs ⇒ rev xs ! i ∈ set xs
  using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)
} note H = this
have ∀ i < length rem. rev rem ! i < card (atms-of-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 μC-bounded[of rev rem card (atms-of-ms A)+2 unassigned-lit A l] T undef-L
  by (simp add: rem μC-append μC-cons F tr-S)
qed

```

lemma *dpll-bj-trail-mes-decreasing-prop*:

assumes *dpll*: *dpll-bj S T* **and** *inv*: *inv S* **and**
N-A: *atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A* **and**
M-A: *atm-of ' lits-of-l (trail S) ⊆ atms-of-ms A* **and**
nd: *no-dup (trail S)* **and**
fin-A: *finite A*

shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $\quad < (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

proof –

```

let ?b = 2 + card (atms-of-ms A)
let ?s = 1 + card (atms-of-ms A)
let ?μ = μC ?s ?b
have M'-A: atm-of ' lits-of-l (trail T) ⊆ 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 ⇒ length xs - Suc i < length xs
  by auto
  then have H: i < length xs ⇒ xs ! i ∈ set xs
  using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)
} note H = this

have l-M-A: length (trail S) ≤ card (atms-of-ms A)
  by (simp add: fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd)
have l-M'-A: length (trail T) ≤ card (atms-of-ms A)
  by (simp add: fin-A M'-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd')
have l-trail-weight-M: length (trail-weight T) ≤ 1 + card (atms-of-ms A)
  using l-M'-A length-get-all-marked-decomposition-length[of trail T] by auto
have bounded-M: ∀ i < length (trail-weight T). (trail-weight T) ! i < card (atms-of-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 $\mu_C \ ?s \ ?b \ (\text{trail-weight } S) < \mu_C \ ?s \ ?b \ (\text{trail-weight } T)$ **by** *simp*
moreover from $\mu_C\text{-bounded}$ [*OF bounded-M l-trail-weight-M*]
have $\mu_C \ ?s \ ?b \ (\text{trail-weight } T) \leq ?b \wedge ?s$ **by** *auto*
ultimately show *?thesis* **by** *linarith*
qed

lemma *wf-dpll-bj*:

assumes *fin*: *finite A*

shows *wf* $\{(T, S). \text{dpll-bj } S \ T$

$\wedge \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } ' \text{ lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

(is *wf* *?A*)

proof (*rule wf-bounded-measure*[*of -*

$\lambda\cdot. (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$

$\lambda S. \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)]$)

fix *a b* :: '*st*

let *?b* = $2 + \text{card } (\text{atms-of-ms } A)$

let *?s* = $1 + \text{card } (\text{atms-of-ms } A)$

let *?μ* = $\mu_C \ ?s \ ?b$

assume *ab*: $(b, a) \in \{(T, S). \text{dpll-bj } S \ T$

$\wedge \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } ' \text{ lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

have *fin-A*: *finite* (*atms-of-ms A*)

using *fin* **by** *auto*

have

dpll-bj: *dpll-bj a b and*

N-A: *atms-of-mm* (*clauses*_{NOT} *a*) $\subseteq \text{atms-of-ms } A$ **and**

M-A: *atm-of* ' *lits-of-l* (*trail a*) $\subseteq \text{atms-of-ms } A$ **and**

nd: *no-dup* (*trail a*) **and**

inv: *inv a*

using *ab* **by** *auto*

have *M'-A*: *atm-of* ' *lits-of-l* (*trail b*) $\subseteq \text{atms-of-ms } A$

by (*meson M-A N-A* $\langle \text{dpll-bj } a \ b \rangle \text{dpll-bj-atms-in-trail-in-set inv}$)

have *nd'*: *no-dup* (*trail b*)

using $\langle \text{dpll-bj } a \ b \rangle \text{dpll-bj-no-dup nd inv}$ **by** *blast*

{ fix *i* :: *nat* **and** *xs* :: '*a list*

have $i < \text{length } xs \implies \text{length } xs - \text{Suc } i < \text{length } xs$

by *auto*

then have *H*: $i < \text{length } xs \implies xs ! i \in \text{set } xs$

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*: $\text{length } (\text{trail } a) \leq \text{card } (\text{atms-of-ms } A)$

by (*simp add: fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd*)

have *l-M'-A*: $\text{length } (\text{trail } b) \leq \text{card } (\text{atms-of-ms } A)$

by (*simp add: fin-A M'-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd'*)

have *l-trail-weight-M*: $\text{length } (\text{trail-weight } b) \leq 1 + \text{card } (\text{atms-of-ms } A)$

using *l-M'-A length-get-all-marked-decomposition-length*[*of trail b*] **by** *auto*

have *bounded-M*: $\forall i < \text{length } (\text{trail-weight } b). (\text{trail-weight } b) ! i < \text{card } (\text{atms-of-ms } A) + 2$

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 $\mu_C \ ?s \ ?b \ (\text{trail-weight } a) < \mu_C \ ?s \ ?b \ (\text{trail-weight } b)$ **by** *simp*
moreover from $\mu_C\text{-bounded}$ [*OF bounded-M l-trail-weight-M*]
have $\mu_C \ ?s \ ?b \ (\text{trail-weight } b) \leq ?b \wedge ?s$ **by** *auto*
ultimately show $?b \wedge ?s \leq ?b \wedge ?s \wedge$
 $\mu_C \ ?s \ ?b \ (\text{trail-weight } b) \leq ?b \wedge ?s \wedge$
 $\mu_C \ ?s \ ?b \ (\text{trail-weight } a) < \mu_C \ ?s \ ?b \ (\text{trail-weight } b)$
by *blast*
qed

15.3.4 Normal Forms

We prove that given a normal form of DPLL, with some invariants, the either N is satisfiable and the built valuation M is a model; or N is unsatisfiable.

Idea of the proof: We have to prove that *satisfiable* N , $\neg M \models_{as} N$ and there is no remaining step is incompatible.

1. The *decide* rules tells us that every variable in N has a value.
2. $\neg M \models_{as} N$ tells us that there is conflict.
3. There is at least one decision in the trail (otherwise, M is a model of N).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj* S

theorem *dpll-backjump-final-state*:

fixes $A :: 'v \text{ literal multiset set}$ **and** $S \ T :: 'st$

assumes

atms-of-mm (*clauses*_{NOT} S) \subseteq *atms-of-ms* A **and**

atm-of ' *lits-of-l* (*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*_{NOT} S) (*get-all-marked-decomposition* (*trail* S))

shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} S))

\vee (*trail* $S \models_{asm}$ *clauses*_{NOT} $S \wedge$ *satisfiable* (*set-mset* (*clauses*_{NOT} S)))

proof –

let $?N = \text{set-mset} \ (\text{clauses}_{NOT} \ S)$

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** $\text{sat} = \text{this}(1)$ **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  $'I \subseteq \text{atms-of-ms } ?N$ 
  using sat unfolding satisfiable-def-min by auto
let  $?I = I \cup \{P \mid P. P \in \text{lits-of-l } ?M \wedge \text{atm-of } P \notin \text{atm-of } 'I\}$ 
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  $\langle \text{no-dup } ?M \rangle$  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  $?I$   $(?N \cup \text{unmark-l } ?M)$ 
  using tot atm-I-N unfolding total-over-m-def total-over-set-def
  by (fastforce simp: image-iff)
have  $\{P \mid P. P \in \text{lits-of-l } ?M \wedge \text{atm-of } P \notin \text{atm-of } 'I\} \models_s ?O$ 
  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  $?I$   $(?N \cup ?O)$ 
  using atm-I-N tot unfolding total-over-m-def total-over-set-def
  by (force simp: image-iff lits-of-def dest!: is-marked-ex-Marked)

have atms-N-M: atms-of-ms  $?N \subseteq \text{atm-of } ' \text{lits-of-l } ?M$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $l :: 'v$  where
     $l-N: l \in \text{atms-of-ms } ?N$  and
     $l-M: l \notin \text{atm-of } ' \text{lits-of-l } ?M$ 
    by auto
  have undefined-lit  $?M$   $(\text{Pos } l)$ 
    using  $l-M$  by (metis Marked-Propagated-in-iff-in-lits-of-l
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
  from  $\text{bj-decide}_{NOT}[\text{OF decide}_{NOT}[\text{OF this}]]$  show False
    using  $l-N$   $n-s$  by (metis literal.sel(1) state-eq_{NOT}-ref)
qed
have  $?M \models_{as} CNot\ C$ 
  by (metis (no-types, lifting)  $\langle C \in \text{set-mset } (\text{clauses}_{NOT}\ S) \rangle \langle \neg \text{trail } S \models_a C \rangle$ 
    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  $\exists l \in \text{set } ?M. \text{is-marked } l$ 
proof (rule ccontr)
  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  $\vartheta[\text{iff}]: \bigwedge I. \text{total-over-m } I \ (?N \cup ?O \cup \text{unmark-l } ?M)$ 
     $\longleftrightarrow \text{total-over-m } I \ (?N \cup \text{unmark-l } ?M)$ 
    unfolding total-over-set-def total-over-m-def atms-of-ms-def by auto
  assume  $\neg ?thesis$ 
  then have  $[\text{simp}]: \{\{\# \text{lit-of } L \# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M\}$ 
     $= \{\{\# \text{lit-of } L \# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M \wedge \text{atm-of } (\text{lit-of } L) \notin \text{atms-of-ms } ?N\}$ 
    by auto
  then have  $?N \cup ?O \models_{ps} \text{unmark-l } ?M$ 
    using all-decomposition-implies-propagated-lits-are-implied  $[\text{OF decomp}]$  by auto

then have  $?I \models_s \text{unmark-l } ?M$ 

```

```

    using cons-I' I'-N tot-I' (I ⊢s ?N ∪ ?O) unfolding ∅ true-clss-clss-def by blast
  then have lits-of-l ?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: ∀ 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 unmark-l ?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 unmark-l ?M
  by auto
have N-M-False: ?N ∪ (λL. {#lit-of L#}) ' (set ?M) ⊢ps {{#}}
  using M (I' ⊢as CNot C) (C ∈ ?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 (no-dup ?M) unfolding M-K by (simp add: defined-lit-map)
moreover
  have ?N ∪ ?C' ⊢ps {{#}}
  proof -
    have A: ?N ∪ ?C' ∪ unmark-l ?M =
      ?N ∪ unmark-l ?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 {a ∈ set (trail S). is-marked a ∧ a ≠ Marked K ()} =
      set (trail S) ∩ {L. is-marked L ∧ L ≠ Marked K ()}
    by auto
  end

```

```

then have tot': total-over-set I
  (atm-of ' lit-of ' (set ?M  $\cap$  {L. is-marked L  $\wedge$  L  $\neq$  Marked K ()}))
using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
{ fix x :: ('v, unit, unit) marked-lit
  assume
    a3: lit-of x  $\notin$  I and
    a1: x  $\in$  set ?M and
    a4: is-marked x and
    a5: x  $\neq$  Marked K ()
  then have Pos (atm-of (lit-of x))  $\in$  I  $\vee$  Neg (atm-of (lit-of x))  $\in$  I
    using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
  moreover have f6: Neg (atm-of (lit-of x)) = - Pos (atm-of (lit-of x))
    by simp
  ultimately have - lit-of x  $\in$  I
    using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1))
} note H = this

have  $\neg I \models_s ?C'$ 
  using  $\langle ?N \cup ?C' \models_{ps} \{\{\#\}\} \rangle$  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  $\models$  image-mset uminus ?C + {#- K#}
  unfolding true-clss-def true-cls-def Bex-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 (image-mset uminus ?C)
  using nm unfolding true-annots-def CNot-def M-K by (auto simp add: lits-of-def)
ultimately have False
  using bj-can-jump[of S F' K F C -K
    image-mset uminus (image-mset lit-of {# L :# mset ?M. is-marked L  $\wedge$  L  $\neq$  Marked K ()#})]
     $\langle C \in ?N \rangle$  n-s  $\langle ?M \models_{as}$  CNot C  $\rangle$  bj-backjump inv  $\langle$  no-dup (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 mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT inv backjump-conds
  propagate-conds
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and

```

```

raw-clauses :: 'st  $\Rightarrow$  'cls and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
inv :: 'st  $\Rightarrow$  bool and
backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  '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 atms-of-mm (clausesNOT  $S$ ) = atms-of-mm (clausesNOT  $T$ )
using assms by (induction rule: rtrancpl-induct)
(auto dest: rtrancpl-dpll-bj-inv dpll-bj-atms-of-ms-clauses-inv)

lemma rtrancpl-dpll-bj-atms-in-trail:
assumes
dpll-bj**  $S T$  and
inv  $S$  and
atm-of ' (lits-of-l (trail  $S$ ))  $\subseteq$  atms-of-mm (clausesNOT  $S$ )
shows atm-of ' (lits-of-l (trail  $T$ ))  $\subseteq$  atms-of-mm (clausesNOT  $T$ )
using assms apply (induction rule: rtrancpl-induct)
using dpll-bj-atms-in-trail dpll-bj-atms-of-ms-clauses-inv rtrancpl-dpll-bj-inv by auto

lemma rtrancpl-dpll-bj-sat-iff:
assumes dpll-bj**  $S T$  and inv  $S$ 
shows  $I \models_{sm} \text{clauses}_{NOT} S \longleftrightarrow I \models_{sm} \text{clauses}_{NOT} T$ 
using assms by (induction rule: rtrancpl-induct)
(auto dest!: dpll-bj-sat-iff simp: rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-atms-in-trail-in-set:
assumes
dpll-bj**  $S T$  and
inv  $S$ 
atms-of-mm (clausesNOT  $S$ )  $\subseteq A$  and
atm-of ' (lits-of-l (trail  $S$ ))  $\subseteq A$ 
shows atm-of ' (lits-of-l (trail  $T$ ))  $\subseteq A$ 
using assms

```

by (induction rule: rtrancpl-induct)
 (auto dest: rtrancpl-dpll-bj-inv
 simp add: dpll-bj-atms-in-trail-in-set rtrancpl-dpll-bj-atms-of-ms-clauses-inv
 rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-all-decomposition-implies-inv:

assumes
 dpll-bj⁺⁺ S T and
 inv S
 all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))
shows all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))
using assms **by** (induction rule: rtrancpl-induct)
 (auto intro: dpll-bj-all-decomposition-implies-inv simp: rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-inv-incl-dpll-bj-inv-trancl:

{(T, S). dpll-bj⁺⁺ S T
 ∧ atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A ∧ atm-of ‘ lits-of-l (trail S) ⊆ atms-of-ms A
 ∧ no-dup (trail S) ∧ inv S}
 ⊆ {(T, S). dpll-bj S T ∧ atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A
 ∧ atm-of ‘ lits-of-l (trail S) ⊆ atms-of-ms A ∧ no-dup (trail S) ∧ inv S}⁺
 (is ?A ⊆ ?B⁺)

proof standard

fix x
assume x-A: x ∈ ?A
obtain S T::'st **where**
 x[simp]: x = (T, S) **by** (cases x) auto
have
 dpll-bj⁺⁺ S T and
 atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A and
 atm-of ‘ lits-of-l (trail S) ⊆ atms-of-ms A and
 no-dup (trail S) and
 inv S
using x-A **by** auto
then show x ∈ ?B⁺ **unfolding** x
proof (induction rule: tranclp-induct)
case base
then show ?case **by** auto
next
case (step T U) **note** step = this(1) and ST = this(2) and IH = this(3)[OF this(4-7)]
 and N-A = this(4) and M-A = this(5) and nd = this(6) and inv = this(7)

have [simp]: atms-of-mm (clauses_{NOT} S) = atms-of-mm (clauses_{NOT} T)
using step rtrancpl-dpll-bj-atms-of-ms-clauses-inv tranclp-into-rtrancpl inv **by** fastforce
have no-dup (trail T)
using local.step nd rtrancpl-dpll-bj-no-dup tranclp-into-rtrancpl inv **by** fastforce
moreover have atm-of ‘ (lits-of-l (trail T)) ⊆ atms-of-ms A
by (metis inv M-A N-A local.step rtrancpl-dpll-bj-atms-in-trail-in-set
 tranclp-into-rtrancpl)
moreover have inv T
using inv local.step rtrancpl-dpll-bj-inv tranclp-into-rtrancpl **by** fastforce
ultimately have (U, T) ∈ ?B **using** ST N-A M-A inv **by** auto
then show ?case **using** IH **by** (rule trancl-into-trancl2)
qed
qed

lemma *wf-tranclp-dpll-bj*:

assumes *fin*: *finite A*

shows *wf* $\{(T, S). \text{dpll-bj}^{++} S T$

$\wedge \text{atms-of-mm} (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } \text{' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

using *wf-trancl*[*OF wf-dpll-bj*[*OF fin*]] *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*

by (*rule wf-subset*)

lemma *dpll-bj-sat-ext-iff*:

dpll-bj S T $\implies \text{inv } S \implies I \models_{\text{sextm}} \text{clauses}_{NOT} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} T$

by (*simp add: dpll-bj-clauses*)

lemma *rtranclp-dpll-bj-sat-ext-iff*:

*dpll-bj** S T* $\implies \text{inv } S \implies I \models_{\text{sextm}} \text{clauses}_{NOT} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} T$

by (*induction rule: rtranclp-induct*) (*simp-all add: rtranclp-dpll-bj-inv dpll-bj-sat-ext-iff*)

theorem *full-dpll-backjump-final-state*:

fixes *A* :: '*v literal multiset set* **and** *S T* :: '*st*

assumes

full: *full dpll-bj S T* **and**

atms-S: *atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A* **and**

atms-trail: *atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A* **and**

n-d: *no-dup (trail S)* **and**

finite A **and**

inv: *inv S* **and**

decomp: *all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))*

shows *unsatisfiable (set-mset (clauses_{NOT} S))*

$\vee (\text{trail } T \models_{\text{asm}} \text{clauses}_{NOT} S \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} S)))$

proof –

have *st*: *dpll-bj** S T* **and** *no-step dpll-bj T*

using *full unfolding full-def* **by** *fast+*

moreover have *atms-of-mm (clauses_{NOT} T) \subseteq atms-of-ms A*

using *atms-S inv rtranclp-dpll-bj-atms-of-ms-clauses-inv st* **by** *blast*

moreover have *atm-of ' lits-of-l (trail T) \subseteq atms-of-ms A*

using *atms-S atms-trail inv rtranclp-dpll-bj-atms-in-trail-in-set st* **by** *auto*

moreover have *no-dup (trail T)*

using *n-d inv rtranclp-dpll-bj-no-dup st* **by** *blast*

moreover have *inv*: *inv T*

using *inv rtranclp-dpll-bj-inv st* **by** *blast*

moreover

have *decomp*: *all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))*

using $\langle \text{inv } S \rangle$ *decomp rtranclp-dpll-bj-all-decomposition-implies-inv st* **by** *blast*

ultimately have *unsatisfiable (set-mset (clauses_{NOT} T))*

$\vee (\text{trail } T \models_{\text{asm}} \text{clauses}_{NOT} T \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} T)))$

using $\langle \text{finite } A \rangle$ *dpll-backjump-final-state* **by** *force*

then show *?thesis*

by (*meson* $\langle \text{inv } S \rangle$ *rtranclp-dpll-bj-sat-iff satisfiable-carac st true-annots-true-cls*)

qed

corollary *full-dpll-backjump-final-state-from-init-state*:

fixes *A* :: '*v literal multiset set* **and** *S T* :: '*st*

assumes

full: *full dpll-bj S T* **and**

trail S = \square **and**

clauses_{NOT} S = *N* **and**

$inv\ S$
shows $unsatisfiable\ (set-mset\ N) \vee (trail\ T \models_{asm} N \wedge satisfiable\ (set-mset\ N))$
using $assms\ full-dpll-backjump-final-state[of\ S\ T\ set-mset\ N]$ **by** $auto$

lemma $trancpl-dpll-bj-trail-mes-decreasing-prop$:
assumes $dpll: dpll-bj^{++}\ S\ T$ **and** $inv: inv\ S$ **and**
 $N-A: atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A$ **and**
 $M-A: atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$ **and**
 $n-d: no-dup\ (trail\ S)$ **and**
 $fin-A: finite\ A$
shows $(2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $\quad - \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ T)$
 $\quad < (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $\quad - \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ S)$
using $dpll$
proof $(induction)$
case $base$
then show $?case$
using $N-A\ M-A\ n-d\ dpll-bj-trail-mes-decreasing-prop\ fin-A\ inv$ **by** $blast$

next
case $(step\ T\ U)$ **note** $st = this(1)$ **and** $dpll = this(2)$ **and** $IH = this(3)$
have $atms-of-mm\ (clauses_{NOT}\ S) = atms-of-mm\ (clauses_{NOT}\ T)$
using $rtrancpl-dpll-bj-atms-of-ms-clauses-inv$ **by** $(metis\ dpll-bj-clauses\ dpll-bj-inv\ inv\ st\ trancplD)$
then have $N-A': atms-of-mm\ (clauses_{NOT}\ T) \subseteq atms-of-ms\ A$
using $N-A$ **by** $auto$
moreover have $M-A': atm-of\ 'lits-of-l\ (trail\ T) \subseteq atms-of-ms\ A$
by $(meson\ M-A\ N-A\ inv\ rtrancpl-dpll-bj-atms-in-trail-in-set\ st\ dpll\ trancpl.r-into-trancpl\ trancpl-into-rtrancpl\ trancpl-trans)$
moreover have $nd: no-dup\ (trail\ T)$
by $(metis\ inv\ n-d\ rtrancpl-dpll-bj-no-dup\ st\ trancpl-into-rtrancpl)$
moreover have $inv\ T$
by $(meson\ dpll\ dpll-bj-inv\ inv\ rtrancpl-dpll-bj-inv\ st\ trancpl-into-rtrancpl)$
ultimately show $?case$
using $IH\ dpll-bj-trail-mes-decreasing-prop[of\ T\ U\ A]\ dpll\ fin-A$ **by** $linarith$

qed

end

15.4 CDCL

15.4.1 Learn and Forget

locale $learn-ops =$
 $dpll-state\ mset-cls\ union-cls\ insert-cls\ remove-lit$
 $mset-clss\ union-clss\ in-clss\ insert-clss\ remove-from-clss$
 $trail\ raw-clauses\ prepend-trail\ tl-trail\ add-cls_{NOT}\ remove-cls_{NOT}$
for
 $mset-cls:: 'cls \Rightarrow 'v\ clause$ **and**
 $union-cls:: 'cls \Rightarrow 'cls \Rightarrow 'cls$ **and**
 $insert-cls:: 'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
 $remove-lit:: 'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
 $mset-clss:: 'clss \Rightarrow 'v\ clauses$ **and**
 $union-clss:: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
 $in-clss:: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**
 $insert-clss:: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

```

remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and
trail :: 'st ⇒ ('v, unit, unit) marked-lits and
raw-clauses :: 'st ⇒ 'clss and
prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
tl-trail :: 'st ⇒ 'st and
add-clsNOT :: 'cls ⇒ 'st ⇒ 'st and
remove-clsNOT :: 'cls ⇒ 'st ⇒ 'st +
fixes
  learn-cond :: 'cls ⇒ 'st ⇒ bool
begin
inductive learn :: 'st ⇒ 'st ⇒ bool where
learnNOT-rule: clausesNOT S ⊨pm mset-cls C ⇒
  atms-of (mset-cls C) ⊆ atms-of-mm (clausesNOT S) ∪ atm-of ' (lits-of-l (trail S)) ⇒
  learn-cond C S ⇒
  T ∼ add-clsNOT C S ⇒
  learn S T
inductive-cases learnNOTE: learn S T

lemma learn-μC-stable:
  assumes learn S T and no-dup (trail S)
  shows μC A B (trail-weight S) = μC A B (trail-weight T)
  using assms by (auto elim: learnNOTE)
end

locale forget-ops =
  dpll-state mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
for
  mset-cls:: 'cls ⇒ 'v clause and
  union-cls :: 'cls ⇒ 'cls ⇒ 'cls and
  insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls and
  remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls and
  mset-clss:: 'clss ⇒ 'v clauses and
  union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
  in-clss :: 'cls ⇒ 'clss ⇒ bool and
  insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  raw-clauses :: 'st ⇒ 'clss and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT :: 'cls ⇒ 'st ⇒ 'st and
  remove-clsNOT :: 'cls ⇒ 'st ⇒ 'st +
fixes
  forget-cond :: 'cls ⇒ 'st ⇒ bool
begin
inductive forgetNOT :: 'st ⇒ 'st ⇒ bool where
forgetNOT:
  removeAll-mset (mset-cls C)(clausesNOT S) ⊨pm mset-cls C ⇒
  forget-cond C S ⇒
  C !∈! raw-clauses S ⇒
  T ∼ remove-clsNOT C S ⇒
  forgetNOT S T
inductive-cases forgetNOTE: 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!: forgetNOTE)
end

locale learn-and-forgetNOT =
  learn-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT learn-cond +
forget-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT forget-cond
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  learn-cond forget-cond :: 'cls  $\Rightarrow$  '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

```

15.4.2 Definition of CDCL

```

locale conflict-driven-clause-learning-ops =
  dpll-with-backjumping-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds +
learn-and-forgetNOT mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT learn-cond
forget-cond
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and

```

```

in-cls :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-cls :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-cls :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
inv :: 'st  $\Rightarrow$  bool and
backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
learn-cond forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

inductive cdclNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
c-dpll-bj: dpll-bj S S'  $\Rightarrow$  cdclNOT S S' |
c-learn: learn S S'  $\Rightarrow$  cdclNOT S S' |
c-forgetNOT: forgetNOT S S'  $\Rightarrow$  cdclNOT S S'

lemma cdclNOT-all-induct[consumes 1, case-names dpll-bj learn forgetNOT]:
fixes S T :: 'st
assumes cdclNOT S T and
dpll:  $\bigwedge T. \text{dpll-bj } S \ T \Rightarrow P \ S \ T$  and
learning:
 $\bigwedge C \ T. \text{clauses}_{\text{NOT}} \ S \models_{\text{pm}} \text{mset-cls } C \Rightarrow$ 
 $\text{atms-of } (\text{mset-cls } C) \subseteq \text{atms-of-mm } (\text{clauses}_{\text{NOT}} \ S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \Rightarrow$ 
 $T \sim \text{add-cls}_{\text{NOT}} \ C \ S \Rightarrow$ 
 $P \ S \ T$  and
forgetting:  $\bigwedge C \ T. \text{removeAll-mset } (\text{mset-cls } C) (\text{clauses}_{\text{NOT}} \ S) \models_{\text{pm}} \text{mset-cls } C \Rightarrow$ 
 $C \ !\in! \text{raw-clauses } S \Rightarrow$ 
 $T \sim \text{remove-cls}_{\text{NOT}} \ C \ S \Rightarrow$ 
 $P \ S \ T$ 
shows P S T
using assms(1) by (induction rule: cdclNOT.induct)
(auto intro: assms(2, 3, 4) elim!: learnNOTE forgetNOTE)+

lemma cdclNOT-no-dup:
assumes
cdclNOT S T and
inv S and
no-dup (trail S)
shows no-dup (trail T)
using assms by (induction rule: cdclNOT-all-induct) (auto intro: dpll-bj-no-dup)

```

Consistency of the trail lemma cdcl_{NOT}-consistent:

```

assumes
cdclNOT S T and
inv S and
no-dup (trail S)
shows consistent-interp (lits-of-l (trail T))
using cdclNOT-no-dup[OF assms] distinct-consistent-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-dup (trail S)*
shows *atms-of-mm (clauses_{NOT} T) \subseteq atms-of-mm (clauses_{NOT} S) \cup atm-of ' (lits-of-l (trail S))*
using *assms by (induction rule: cdcl_{NOT}-all-induct)*
(auto dest!: dpll-bj-atms-of-ms-clauses-inv set-mp simp add: atms-of-ms-def Union-eq)

lemma *cdcl_{NOT}-atms-in-trail*:

assumes *cdcl_{NOT} S T and inv S and no-dup (trail S)*
and *atm-of ' (lits-of-l (trail S)) \subseteq atms-of-mm (clauses_{NOT} S)*
shows *atm-of ' (lits-of-l (trail T)) \subseteq atms-of-mm (clauses_{NOT} S)*
using *assms by (induction rule: cdcl_{NOT}-all-induct) (auto simp add: dpll-bj-atms-in-trail)*

lemma *cdcl_{NOT}-atms-in-trail-in-set*:

assumes
cdcl_{NOT} S T and inv S and no-dup (trail S) and
atms-of-mm (clauses_{NOT} S) \subseteq A and
atm-of ' (lits-of-l (trail S)) \subseteq A
shows *atm-of ' (lits-of-l (trail T)) \subseteq A*
using *assms*
by *(induction rule: cdcl_{NOT}-all-induct)*
(simp-all add: dpll-bj-atms-in-trail-in-set dpll-bj-atms-of-ms-clauses-inv)

lemma *cdcl_{NOT}-all-decomposition-implies*:

assumes *cdcl_{NOT} S T and inv S and n-d[simp]: no-dup (trail S) and*
all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))
shows
all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))
using *assms(1,2,4)*

proof *(induction rule: cdcl_{NOT}-all-induct)*

case *dpll-bj*

then show *?case*

using *dpll-bj-all-decomposition-implies-inv n-d by blast*

next

case *learn*

then show *?case by (auto simp add: all-decomposition-implies-def)*

next

case *(forget_{NOT} C T) note cls-C = this(1) and C = this(2) and T = this(3) and iniv = this(4)*

and

decomp = this(5)

show *?case*

unfolding *all-decomposition-implies-def Ball-def*

proof *(intro allI, clarify)*

fix *a b*

assume *(a, b) \in set (get-all-marked-decomposition (trail T))*

then have *unmark-l a \cup set-mset (clauses_{NOT} S) \models ps unmark-l b*

using *decomp T by (auto simp add: all-decomposition-implies-def)*

moreover

have *mset-cls C \in set-mset (clauses_{NOT} S)*

using *C by blast*

then have *set-mset (clauses_{NOT} T) \models ps set-mset (clauses_{NOT} S)*

by *(metis (no-types, lifting) T clauses-remove-cls_{NOT} cls-C insert-Diff order-refl
set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses true-clss-clss-def
true-clss-clss-insert)*

ultimately show *unmark-l a \cup set-mset (clauses_{NOT} T)*

```

       $\models_{ps} \text{unmark-}l \ b$ 
      using true-clss-clss-generalise-true-clss-clss by blast
    qed
  qed

```

Extension of models lemma *cdcl_{NOT}-bj-sat-ext-iff*:

```

  assumes cdclNOT  $S$  T and inv  $S$  and n-d: no-dup (trail  $S$ )
  shows  $I \models_{sextm} \text{clauses}_{NOT} S \longleftrightarrow I \models_{sextm} \text{clauses}_{NOT} T$ 
  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  $T = \text{this}(3)$ 

```

```

  { fix  $J$ 

```

```

    assume

```

```

       $I \models_{sextm} \text{clauses}_{NOT} S$  and

```

```

       $I \subseteq J$  and

```

```

      tot: total-over-m  $J$  (set-mset ( $\{\#mset\text{-cls } C\} + \text{clauses}_{NOT} S$ )) and

```

```

      cons: consistent-interp  $J$ 

```

```

  then have  $J \models_{sm} \text{clauses}_{NOT} S$  unfolding true-clss-ext-def by auto

```

moreover

```

  with  $\langle \text{clauses}_{NOT} S \models_{pm} mset\text{-cls } C \rangle$  have  $J \models mset\text{-cls } C$ 

```

```

  using tot cons unfolding true-clss-clss-def by auto

```

```

  ultimately have  $J \models_{sm} \{\#mset\text{-cls } C\} + \text{clauses}_{NOT} S$  by auto

```

```

}

```

```

then have  $H: I \models_{sextm} (\text{clauses}_{NOT} S) \implies I \models_{sext} \text{insert } (mset\text{-cls } C) (\text{set-mset } (\text{clauses}_{NOT} S))$ 

```

```

  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- $C = \text{this}(1)$  and  $T = \text{this}(3)$ 

```

```

  { fix  $J$ 

```

```

    assume

```

```

       $I \models_{sext} \text{set-mset } (\text{clauses}_{NOT} S) - \{mset\text{-cls } C\}$  and

```

```

       $I \subseteq J$  and

```

```

      tot: total-over-m  $J$  (set-mset ( $\text{clauses}_{NOT} S$ )) and

```

```

      cons: consistent-interp  $J$ 

```

```

  then have  $J \models_s \text{set-mset } (\text{clauses}_{NOT} S) - \{mset\text{-cls } C\}$ 

```

```

  unfolding true-clss-ext-def by (meson Diff-subset total-over-m-subset)

```

moreover

```

  with cls- $C$  have  $J \models mset\text{-cls } C$ 

```

```

  using tot cons unfolding true-clss-clss-def

```

```

  by (metis Un-commute forgetNOT.hyps(2) in-clss-mset-clss insert-Diff insert-is-Un order-refl
      set-mset-minus-replicate-mset(1))

```

```

  ultimately have  $J \models_{sm} (\text{clauses}_{NOT} S)$  by (metis insert-Diff-single true-clss-insert)

```

```

}

```

```

then have  $H: I \models_{sext} \text{set-mset } (\text{clauses}_{NOT} S) - \{mset\text{-cls } C\} \implies I \models_{sextm} (\text{clauses}_{NOT} S)$ 

```

```

  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*

15.5 CDCL with invariant

locale *conflict-driven-clause-learning* =
conflict-driven-clause-learning-ops +
assumes $cdcl_{NOT-inv}$: $\bigwedge S T. cdcl_{NOT} S T \implies inv S \implies inv T$
begin
sublocale *dpll-with-backjumping*
apply *unfold-locales*
using $cdcl_{NOT.simps}$ $cdcl_{NOT-inv}$ **by** auto

lemma *rtranclp-cdcl_{NOT-inv}*:
 $cdcl_{NOT}^{**} S T \implies inv S \implies inv T$
by (induction rule: *rtranclp-induct*) (auto simp add: $cdcl_{NOT-inv}$)

lemma *rtranclp-cdcl_{NOT-no-dup}*:
assumes $cdcl_{NOT}^{**} S T$ **and** $inv S$
and *no-dup* (trail S)
shows *no-dup* (trail T)
using *assms* **by** (induction rule: *rtranclp-induct*) (auto intro: $cdcl_{NOT-no-dup}$ *rtranclp-cdcl_{NOT-inv}*)

lemma *rtranclp-cdcl_{NOT-trail-clauses-bound}*:
assumes
 $cdcl$: $cdcl_{NOT}^{**} S T$ **and**
 inv : $inv S$ **and**
 $n-d$: *no-dup* (trail S) **and**
 $atms-clauses-S$: $atms-of-mm (clauses_{NOT} S) \subseteq A$ **and**
 $atms-trail-S$: $atm-of (lits-of-l (trail S)) \subseteq A$
shows $atm-of (lits-of-l (trail T)) \subseteq A \wedge atms-of-mm (clauses_{NOT} T) \subseteq A$
using $cdcl$
proof (induction rule: *rtranclp-induct*)
case *base*
then show ?case **using** $atms-clauses-S$ $atms-trail-S$ **by** *simp*
next
case (step $T U$) **note** $st = this(1)$ **and** $cdcl_{NOT} = this(2)$ **and** $IH = this(3)$
have $inv T$ **using** $inv st$ *rtranclp-cdcl_{NOT-inv}* **by** *blast*
have *no-dup* (trail T)
using *rtranclp-cdcl_{NOT-no-dup}*[of $S T$] st $cdcl_{NOT}$ inv $n-d$ **by** *blast*
then have $atms-of-mm (clauses_{NOT} U) \subseteq A$
using $cdcl_{NOT-atms-of-ms-clauses-decreasing}$ [OF $cdcl_{NOT}$] IH $n-d$ $\langle inv T \rangle$ **by** *fast*
moreover
have $atm-of (lits-of-l (trail U)) \subseteq A$
using $cdcl_{NOT-atms-in-trail-in-set}$ [OF $cdcl_{NOT}$, of A] $\langle no-dup (trail T) \rangle$
by (meson $atms-trail-S$ $atms-clauses-S$ IH $\langle inv T \rangle$ $cdcl_{NOT}$)
ultimately show ?case **by** *fast*
qed

lemma *rtranclp-cdcl_{NOT-all-decomposition-implies}*:
assumes $cdcl_{NOT}^{**} S T$ **and** $inv S$ **and** *no-dup* (trail S) **and**
 $all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))$
shows
 $all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))$

using *assms* **by** (*induction*)
 (*auto intro: rtrancpl-cdcl_{NOT}-inv cdcl_{NOT}-all-decomposition-implies rtrancpl-cdcl_{NOT}-no-dup*)

lemma *rtrancpl-cdcl_{NOT}-bj-sat-ext-iff*:
assumes *cdcl_{NOT}** S T and inv S and no-dup (trail S)*
shows $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T$
using *assms* **apply** (*induction rule: rtrancpl-induct*)
using *cdcl_{NOT}-bj-sat-ext-iff* **by** (*auto intro: rtrancpl-cdcl_{NOT}-inv rtrancpl-cdcl_{NOT}-no-dup*)

definition *cdcl_{NOT}-NOT-all-inv* **where**
 $\text{cdcl}_{\text{NOT}}\text{-NOT-all-inv } A \ S \longleftrightarrow (\text{finite } A \wedge \text{inv } S \wedge \text{atms-of-mm } (\text{clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{no-dup } (\text{trail } S))$

lemma *cdcl_{NOT}-NOT-all-inv*:
assumes *cdcl_{NOT}** S T and cdcl_{NOT}-NOT-all-inv A S*
shows *cdcl_{NOT}-NOT-all-inv A T*
using *assms* **unfolding** *cdcl_{NOT}-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**
 $\text{learn-or-forget } S \ T \equiv (\lambda S \ T. \text{learn } S \ T \vee \text{forget}_{\text{NOT}} S \ T) \ S \ T$

lemma *rtrancpl-learn-or-forget-cdcl_{NOT}*:
 $\text{learn-or-forget}^{**} S \ T \implies \text{cdcl}_{\text{NOT}}^{**} S \ T$
using *rtrancpl-mono[of learn-or-forget cdcl_{NOT}] cdcl_{NOT}.c-learn cdcl_{NOT}.c-forget_{NOT}* **by** *blast*

lemma *learn-or-forget-dpll- μ_C* :
assumes
*l-f: learn-or-forget** S T and*
dpll: dpll-bj T U and
inv: cdcl_{NOT}-NOT-all-inv A S
shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } U)$
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$
 $(\text{is } ?\mu \ U < ?\mu \ S)$

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

$\bigwedge i. cdcl_{NOT} (f i) (f (Suc i))$ **and**

$inv: cdcl_{NOT-NOT-all-inv} A (f 0)$

shows $\exists j. \forall i \geq j. learn-or-forget (f i) (f (Suc i))$

using *assms*

proof (*induction* $(2 + card (atms-of-ms A)) \wedge (1 + card (atms-of-ms A))$)

$- \mu_C (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight (f 0))$

arbitrary: f

rule: nat-less-induct-case

case $(Suc n)$ **note** $IH = this(1)$ **and** $\mu = this(2)$ **and** $cdcl_{NOT} = this(3)$ **and** $inv = this(4)$

consider

$(dpll-end) \exists j. \forall i \geq j. learn-or-forget (f i) (f (Suc i))$

$| (dpll-more) \neg(\exists j. \forall i \geq j. learn-or-forget (f i) (f (Suc i)))$

by *blast*

then show *?case*

proof *cases*

case *dpll-end*

then show *?thesis* **by** *auto*

next

case *dpll-more*

then have $j: \exists i. \neg learn (f i) (f (Suc i)) \wedge \neg forget_{NOT} (f i) (f (Suc i))$

by *blast*

obtain *i* **where**

$\neg learn (f i) (f (Suc i)) \wedge \neg forget_{NOT} (f i) (f (Suc i))$ **and**

$\forall k < i. learn-or-forget (f k) (f (Suc k))$

proof $-$

obtain i_0 **where** $\neg learn (f i_0) (f (Suc i_0)) \wedge \neg forget_{NOT} (f i_0) (f (Suc i_0))$

using *j* **by** *auto*

then have $\{i. i \leq i_0 \wedge \neg learn (f i) (f (Suc i)) \wedge \neg forget_{NOT} (f i) (f (Suc i))\} \neq \{\}$

by *auto*

let $?I = \{i. i \leq i_0 \wedge \neg learn (f i) (f (Suc i)) \wedge \neg forget_{NOT} (f i) (f (Suc i))\}$

let $?i = Min ?I$

have *finite ?I*

by *auto*

have $\neg learn (f ?i) (f (Suc ?i)) \wedge \neg forget_{NOT} (f ?i) (f (Suc ?i))$

using *Min-in[OF (finite ?I) (?I $\neq \{\}$)]* **by** *auto*

moreover have $\forall k < ?i. learn-or-forget (f k) (f (Suc k))$

using *Min.coboundedI[of {i. i $\leq i_0 \wedge \neg learn (f i) (f (Suc i)) \wedge \neg forget_{NOT} (f i) (f (Suc i))$ }, simplified]*

by (*meson* $\neg learn (f i_0) (f (Suc i_0)) \wedge \neg forget_{NOT} (f i_0) (f (Suc i_0))$) *less-imp-le dual-order.trans not-le*

ultimately show *?thesis* **using** *that* **by** *blast*

qed

def $g \equiv \lambda n. f (n + Suc i)$

have $dpll-bj (f i) (g 0)$

using $\neg learn (f i) (f (Suc i)) \wedge \neg forget_{NOT} (f i) (f (Suc i))$ *cdcl_{NOT} cdcl_{NOT}.cases*

g-def **by** *auto*

{

fix *j*

assume $j \leq i$

then have $learn-or-forget^{**} (f 0) (f j)$

```

    apply (induction j)
    apply simp
    by (metis (no-types, lifting) Suc-leD Suc-le-lessD rtranclp.simps
         $\langle \forall k < i. \text{learn } (f k) (f (Suc k)) \vee \text{forget}_{NOT} (f k) (f (Suc k)) \rangle$ )
  }
  then have learn-or-forget** (f 0) (f i) by blast
  then have (2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
    -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight (g 0))
    < (2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
    -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight (f 0))
  using learn-or-forget-dpll- $\mu_C$ [of f 0 f i g 0 A] inv  $\langle \text{dpll-bj } (f i) (g 0) \rangle$ 
  unfolding cdclNOT-NOT-all-inv-def by linarith

  moreover have cdclNOT-i: cdclNOT** (f 0) (g 0)
  using rtranclp-learn-or-forget-cdclNOT[of f 0 f i]  $\langle \text{learn-or-forget** } (f 0) (f i) \rangle$ 
  cdclNOT[of i] unfolding g-def by auto
  moreover have  $\bigwedge i. \text{cdcl}_{NOT} (g i) (g (Suc i))$ 
  using cdclNOT g-def by auto
  moreover have cdclNOT-NOT-all-inv A (g 0)
  using inv cdclNOT-i rtranclp-cdclNOT-trail-clauses-bound g-def cdclNOT-NOT-all-inv by auto
  ultimately obtain j where j:  $\bigwedge i. i \geq j \implies \text{learn-or-forget } (g i) (g (Suc i))$ 
  using IH unfolding  $\mu$ [symmetric] by presburger
  show ?thesis
  proof
    {
      fix k
      assume k  $\geq j + \text{Suc } i$ 
      then have learn-or-forget (f k) (f (Suc k))
        using j[of k-Suc i] unfolding g-def by auto
    }
    then show  $\forall k \geq j + \text{Suc } i. \text{learn-or-forget } (f k) (f (Suc k))$ 
    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:  $\exists i. \neg \text{learn } (f i) (f (Suc i)) \wedge \neg \text{forget}_{NOT} (f i) (f (Suc i))$ 
  by blast
  obtain i where
     $\neg \text{learn } (f i) (f (Suc i)) \wedge \neg \text{forget}_{NOT} (f i) (f (Suc i))$  and
     $\forall k < i. \text{learn-or-forget } (f k) (f (Suc k))$ 
  proof -
    obtain i0 where  $\neg \text{learn } (f i_0) (f (Suc i_0)) \wedge \neg \text{forget}_{NOT} (f i_0) (f (Suc i_0))$ 
    using j by auto
    then have  $\{i. i \leq i_0 \wedge \neg \text{learn } (f i) (f (Suc i)) \wedge \neg \text{forget}_{NOT} (f i) (f (Suc i))\} \neq \{\}$ 
    by auto
    let ?I =  $\{i. i \leq i_0 \wedge \neg \text{learn } (f i) (f (Suc i)) \wedge \neg \text{forget}_{NOT} (f i) (f (Suc i))\}$ 
    let ?i = Min ?I
    have finite ?I
    by auto
    have  $\neg \text{learn } (f ?i) (f (Suc ?i)) \wedge \neg \text{forget}_{NOT} (f ?i) (f (Suc ?i))$ 
    using Min-in[OF finite ?I  $\langle ?I \neq \{\} \rangle$ ] by auto
  end

```



```

moreover have  $\forall k < ?i. \text{learn-or-forget } (f\ k) (f\ (\text{Suc } k))$ 
using Min.coboundedI[of  $\{i. i \leq i_0 \wedge \neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))\}$ , simplified]
by (meson  $\neg \text{learn } (f\ i_0) (f\ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f\ i_0) (f\ (\text{Suc } i_0))$ ) less-imp-le
dual-order.trans not-le
ultimately show ?thesis using that by blast
qed
have dpll-bj  $(f\ i) (f\ (\text{Suc } i))$ 
using  $\neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f\ i) (f\ (\text{Suc } i))$  cdclNOT cdclNOT.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  $f\ 0\ f\ i\ f\ (\text{Suc } i)\ A$ ] inv 0
 $\langle \text{dpll-bj } (f\ i) (f\ (\text{Suc } i)) \rangle$  unfolding cdclNOT-NOT-all-inv-def by linarith
qed
qed

```

lemma *wf-cdcl_{NOT}-no-learn-and-forget-infinite-chain*:

```

assumes
  no-infinite-lf:  $\bigwedge f\ j. \neg (\forall i \geq j. \text{learn-or-forget } (f\ i) (f\ (\text{Suc } i)))$ 
shows wf  $\{(T, S). \text{cdcl}_{NOT} S\ T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A\ S\}$  (is wf  $\{(T, S). \text{cdcl}_{NOT} S\ T \wedge ?\text{inv } S\}$ )
unfolding wf-iff-no-infinite-down-chain
proof (rule ccontr)
assume  $\neg \neg (\exists f. \forall i. (f\ (\text{Suc } i), f\ i) \in \{(T, S). \text{cdcl}_{NOT} S\ T \wedge ?\text{inv } S\})$ 
then obtain f where
   $\forall i. \text{cdcl}_{NOT} (f\ i) (f\ (\text{Suc } i)) \wedge ?\text{inv } (f\ i)$ 
by fast
then have  $\exists j. \forall i \geq j. \text{learn-or-forget } (f\ i) (f\ (\text{Suc } i))$ 
using infinite-cdclNOT-exists-learn-and-forget-infinite-chain[of f] by meson
then show False using no-infinite-lf by blast
qed

```

lemma *inv-and-tranclp-cdcl_{NOT}-tranclp-cdcl_{NOT}-and-inv*:

```

 $\text{cdcl}_{NOT}^{++} S\ T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A\ S \longleftrightarrow (\lambda S\ T. \text{cdcl}_{NOT} S\ T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A\ S)^{++} S\ T$ 
(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) cdclNOT-NOT-all-inv tranclp.simps tranclp-into-rtranclp)

```

next

assume ?B
 then have ?A by induction auto
 moreover have ?I using ⟨?B⟩ tranclpD by fastforce
 ultimately show ?A ∧ ?I by blast
 qed

lemma *wf-tranclp-cdcl_{NOT}-no-learn-and-forget-infinite-chain*:

assumes
 no-infinite-lf: $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i)))$
 shows *wf* $\{(T, S). \text{cdcl}_{NOT}^{++} S \ T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A \ S\}$
 using *wf-trancl*[OF *wf-cdcl_{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* *cdcl_{NOT}* *S* and
 inv: *cdcl_{NOT}-NOT-all-inv* *A* *S* and
 decomp: *all-decomposition-implies-m* (*clauses_{NOT}* *S*) (*get-all-marked-decomposition* (*trail* *S*))
 shows *unsatisfiable* (*set-mset* (*clauses_{NOT}* *S*))
 $\vee (\text{trail } S \models_{asm} \text{clauses}_{NOT} S \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} S)))$

proof —

have *n-s'*: *no-step* *dpll-bj* *S*
 using *n-s* by (auto simp: *cdcl_{NOT}.sims*)
 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_{NOT}* *S*) (*get-all-marked-decomposition* (*trail* *S*))
 shows *unsatisfiable* (*set-mset* (*clauses_{NOT}* *T*))
 $\vee (\text{trail } T \models_{asm} \text{clauses}_{NOT} T \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} T)))$

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_{NOT}* *T*) (*get-all-marked-decomposition* (*trail* *T*))
 using *cdcl_{NOT}-NOT-all-inv-def* *decomp* *inv* *rtranclp-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*

15.6 Termination

15.6.1 Restricting learn and forget

locale *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =
 dpll-state *mset-cls* *union-cls* *insert-cls* *remove-lit*

mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-clss_{NOT} remove-clss_{NOT} +
conflict-driven-clause-learning mset-clss union-clss insert-clss remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-clss_{NOT} remove-clss_{NOT}
inv backjump-conds propagate-conds
 $\lambda C S. \text{distinct-mset } (mset\text{-cls } C) \wedge \neg \text{tautology } (mset\text{-cls } C) \wedge \text{learn-restrictions } C S \wedge$
 $(\exists F K d F' C' L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge mset\text{-cls } C = C' + \{\#L\# \} \wedge F \models_{as} CNot$
 C'
 $\wedge C' + \{\#L\# \} \notin \# \text{clauses}_{NOT} S)$
 $\lambda C S. \neg(\exists F' F K d L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} CNot (\text{remove1-mset } L (mset\text{-cls}$
 $C)))$
 $\wedge \text{forget-restrictions } C S$
for
mset-clss :: 'cls \Rightarrow 'v clause and
union-clss :: 'cls \Rightarrow 'cls \Rightarrow 'cls and
insert-clss :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and
remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and
mset-clss :: 'clss \Rightarrow 'v clauses and
union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss and
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool and
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and
remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and
trail :: 'st \Rightarrow ('v, unit, unit) marked-lits and
raw-clauses :: 'st \Rightarrow 'clss and
prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st and
tl-trail :: 'st \Rightarrow 'st and
add-clss_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st and
remove-clss_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st and
inv :: 'st \Rightarrow bool and
backjump-conds :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool and
propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool and
learn-restrictions forget-restrictions :: 'cls \Rightarrow 'st \Rightarrow bool

begin

lemma *cdcl_{NOT}-learn-all-induct*[*consumes 1, case-names dp11-bj learn forget_{NOT}*]:

fixes *S T :: 'st*

assumes *cdcl_{NOT} S T and*

dp11: $\bigwedge T. dp11\text{-bj } S T \implies P S T$ and

learning:

$\bigwedge C F K F' C' L T. \text{clauses}_{NOT} S \models_{pm} mset\text{-cls } C \implies$
 $\text{atms-of } (mset\text{-cls } C) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \implies$
 $\text{distinct-mset } (mset\text{-cls } C) \implies$
 $\neg \text{tautology } (mset\text{-cls } C) \implies$
 $\text{learn-restrictions } C S \implies$
 $\text{trail } S = F' @ \text{Marked } K () \# F \implies$
 $mset\text{-cls } C = C' + \{\#L\# \} \implies$
 $F \models_{as} CNot C' \implies$
 $C' + \{\#L\# \} \notin \# \text{clauses}_{NOT} S \implies$
 $T \sim \text{add-clss}_{NOT} C S \implies$
 $P S T$ **and**

forgetting: $\bigwedge C T. \text{removeAll-mset } (mset\text{-cls } C) (\text{clauses}_{NOT} S) \models_{pm} mset\text{-cls } C \implies$

$C ! \in ! \text{raw-clauses } S \implies$

$\neg(\exists F' F K L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} CNot (mset\text{-cls } C - \{\#L\# \})) \implies$

$T \sim \text{remove-clss}_{NOT} C S \implies$

```

forget-restrictions C S  $\implies$ 
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 *rtrancp-cdcl_{NOT}-inv*:
 $cdcl_{NOT}^{**} S T \implies inv S \implies inv T$
apply (induction rule: rtrancp-induct)
apply simp
using *cdcl_{NOT}-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 (clausesNOT T - clausesNOT S)
   $\subseteq$  simple-clss (atms-of-mm (clausesNOT S)  $\cup$  atm-of ' lits-of-l (trail S))

```

proof

```

fix C assume C: C  $\in$  set-mset (clausesNOT T - clausesNOT S)
have distinct-mset C  $\neg$ tautology C using learn C n-d by (elim learnNOTE; auto)+
then have C  $\in$  simple-clss (atms-of C)
  using distinct-mset-not-tautology-implies-in-simple-clss by blast
moreover have atms-of C  $\subseteq$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' lits-of-l (trail S)
  using learn C n-d by (elim learnNOTE) (auto simp: atms-of-ms-def atms-of-def image-Un
    true-annots-CNot-all-atms-defined)
moreover have finite (atms-of-mm (clausesNOT S)  $\cup$  atm-of ' lits-of-l (trail S))
  by auto
ultimately show C  $\in$  simple-clss (atms-of-mm (clausesNOT S)  $\cup$  atm-of ' lits-of-l (trail S))
  using simple-clss-mono by (metis (no-types) insert-subset mk-disjoint-insert)

```

qed

definition *conflicting-bj-clss* S \equiv

```

{C + {#L#} | C L. C + {#L#}  $\in$  # clausesNOT S  $\wedge$  distinct-mset (C + {#L#})
 $\wedge$   $\neg$ tautology (C + {#L#})
 $\wedge$  ( $\exists F' K F$ . trail S = F' @ Marked K () # F  $\wedge$  F  $\models_{as}$  CNot C)}

```

lemma *conflicting-bj-clss-remove-cl_{NOT}[simp]*:

```

conflicting-bj-clss (remove-clNOT C S) = conflicting-bj-clss S - {mset-clNOT C}
unfolding conflicting-bj-clss-def by fastforce

```

lemma *conflicting-bj-clss-remove-cl_{NOT}'[simp]*:

```

T  $\sim$  remove-clNOT C S  $\implies$  conflicting-bj-clss T = conflicting-bj-clss S - {mset-clNOT C}
unfolding conflicting-bj-clss-def by fastforce

```

lemma *conflicting-bj-clss-add-cl_{NOT}-state-eq*:

```

T  $\sim$  add-clNOT C' S  $\implies$  no-dup (trail S)  $\implies$  conflicting-bj-clss T
= conflicting-bj-clss S
 $\cup$  (if  $\exists C L$ . mset-clNOT C' = C + {#L#}  $\wedge$  distinct-mset (C + {#L#})  $\wedge$   $\neg$ tautology (C + {#L#})
 $\wedge$  ( $\exists F' K d F$ . trail S = F' @ Marked K () # F  $\wedge$  F  $\models_{as}$  CNot C))

```

$then \{mset-cls\ C'\} \text{ else } \{\}$
unfolding *conflicting-bj-clss-def* **by** *metis+*

lemma *conflicting-bj-clss-add-clss_{NOT}*:
 $no-dup\ (trail\ S) \implies$
 $conflicting-bj-clss\ (add-clss_{NOT}\ C'\ S)$
 $=\ conflicting-bj-clss\ S$
 $\cup\ (if\ \exists\ C\ L.\ mset-cls\ C' = C + \{\#L\# \} \wedge distinct-mset\ (C + \{\#L\# \}) \wedge \neg tautology\ (C + \{\#L\# \})$
 $\wedge\ (\exists\ F'\ K\ d\ F.\ trail\ S = F' @ Marked\ K\ ()\ \# F \wedge F \models_{as}\ CNot\ C)$
 $then\ \{mset-cls\ C'\} \text{ else } \{\}$
using *conflicting-bj-clss-add-clss_{NOT}-state-eq* **by** *auto*

lemma *conflicting-bj-clss-incl-clauses*:
 $conflicting-bj-clss\ S \subseteq set-mset\ (clauses_{NOT}\ S)$
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 $(elim\ learn_{NOT}\ E)$
by $(subst\ conflicting-bj-clss-add-clss_{NOT}-state-eq[of\ T])\ auto$

abbreviation *conflicting-bj-clss-yet* $b\ S \equiv$
 $3 \wedge b - card\ (conflicting-bj-clss\ S)$

abbreviation $\mu_L :: nat \Rightarrow 'st \Rightarrow nat \times nat$ **where**
 $\mu_L\ b\ S \equiv (conflicting-bj-clss-yet\ b\ S,\ card\ (set-mset\ (clauses_{NOT}\ S)))$

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** $(elim\ forget_{NOT}\ E)$
apply *auto*
unfolding *conflicting-bj-clss-def*
apply *clarify*
using *diff-union-cancelR* **by** $(metis\ diff-union-cancelR)$

lemma *size-mset-removeAll-mset-le-iff*:
 $size\ (removeAll-mset\ x\ M) < size\ M \longleftrightarrow x \in \# M$
apply $(rule\ iffI)$
apply $(force\ intro:\ count-inI)$
apply $(rule\ mset-less-size)$
apply $(auto\ dest:\ simp:\ subset-mset-def\ multiset-eq-iff)$
done

lemma *forget- μ_L -decrease*:
assumes $forget_{NOT}\ S\ T$
shows $(\mu_L\ b\ T,\ \mu_L\ b\ S) \in less-than\ <lex*>\ less-than$
proof –
have $card\ (set-mset\ (clauses_{NOT}\ S)) > 0$
using $forget_{NOT}$ **by** $(elim\ forget_{NOT}\ E)\ (auto\ simp:\ size-mset-removeAll-mset-le-iff\ card-gt-0-iff)$
then have $card\ (set-mset\ (clauses_{NOT}\ T)) < card\ (set-mset\ (clauses_{NOT}\ S))$

using *forget_{NOT}* by (elim *forget_{NOT}E*) (auto simp: *size-mset-removeAll-mset-le-iff*)
 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-neg*:

$A \neq \text{insert } a \ A \longleftrightarrow a \notin A$
 by auto

lemma *learn- μ_L -decrease*:

assumes *learnST*: *learn S T* **and** *n-d*: *no-dup (trail S)* **and**
A: *atms-of-mm (clauses_{NOT} S) \cup atm-of ' lits-of-l (trail S) \subseteq A* **and**
fin-A: *finite A*

shows $(\mu_L (\text{card } A) \ T, \mu_L (\text{card } A) \ S) \in \text{less-than } <*\text{lex}*> \text{less-than}$

proof –

have [*simp*]: $(\text{atms-of-mm (clauses}_{NOT} \ T) \cup \text{atm-of ' lits-of-l (trail } T))$
 $= (\text{atms-of-mm (clauses}_{NOT} \ S) \cup \text{atm-of ' lits-of-l (trail } S))$
using *learnST n-d* **by** (elim *learn_{NOT}E*) auto

then have $\text{card (atms-of-mm (clauses}_{NOT} \ T) \cup \text{atm-of ' lits-of-l (trail } T))}$
 $= \text{card (atms-of-mm (clauses}_{NOT} \ S) \cup \text{atm-of ' lits-of-l (trail } S))}$
by (auto intro!: *card-mono*)

then have $3: (3::\text{nat}) \wedge \text{card (atms-of-mm (clauses}_{NOT} \ T) \cup \text{atm-of ' lits-of-l (trail } T))}$
 $= 3 \wedge \text{card (atms-of-mm (clauses}_{NOT} \ S) \cup \text{atm-of ' lits-of-l (trail } S))}$
by (auto intro: *power-mono*)

moreover have *conflicting-bj-clss S \subseteq conflicting-bj-clss T*
using *learnST n-d* **by** (simp add: *learn-conflicting-increasing*)

moreover have *conflicting-bj-clss S \neq conflicting-bj-clss T*
using *learnST*

proof (elim *learn_{NOT}E*, *goal-cases*)

case (1 *C*) **note** *clss-S = this(1)* **and** *atms-C = this(2)* **and** *inv = this(3)* **and** *T = this(4)*

then obtain *F K F' C' L* **where**

tr-S: *trail S = F' @ Marked K () # F* **and**

C: *mset-cls C = C' + {\#L\#}* **and**

F: *F $\models_{as} C \text{Not } C'$* **and**

C-S: *C' + {\#L\#} \notin clauses_{NOT} S*

by *blast*

moreover have *distinct-mset (mset-cls C) \neg tautology (mset-cls C)* **using** *inv* **by** *blast+*

ultimately have *C' + {\#L\#} \in conflicting-bj-clss T*

using *T n-d* **unfolding** *conflicting-bj-clss-def* **by** *fastforce*

moreover have *C' + {\#L\#} \notin 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-cls_{NOT}*)

ultimately have $\text{card (conflicting-bj-clss } T) \geq \text{card (conflicting-bj-clss } S)$

using *card-mono* **by** *blast*

moreover

```

have fin': finite (atms-of-mm (clausesNOT T)  $\cup$  atm-of ' lits-of-l (trail T))
  by auto
have 1:atms-of-ms (conflicting-bj-clss T)  $\subseteq$  atms-of-mm (clausesNOT 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
 $\subseteq$  simple-clss (atms-of-mm (clausesNOT T)  $\cup$  atm-of ' lits-of-l (trail T))
  by standard (meson 1 2 fin'  $\langle$ finite (conflicting-bj-clss T) $\rangle$  simple-clss-mono
    distinct-mset-set-def simplified-in-simple-clss subsetCE sup.coboundedI1)
moreover
then have #: 3  $\wedge$  card (atms-of-mm (clausesNOT T)  $\cup$  atm-of ' lits-of-l (trail T))
   $\geq$  card (conflicting-bj-clss T)
  by (meson Nat.le-trans simple-clss-card simple-clss-finite card-mono fin')
have atms-of-mm (clausesNOT T)  $\cup$  atm-of ' lits-of-l (trail T)  $\subseteq$  A
  using learnNOTE[OF learnST] A by simp
then have 3  $\wedge$  (card A)  $\geq$  card (conflicting-bj-clss T)
  using # fin-A by (meson simple-clss-card simple-clss-finite
    simple-clss-mono calculation(2) card-mono dual-order.trans)
ultimately show ?thesis
  using psubset-card-mono[OF fin-T ]
  unfolding less-than-iff lex-prod-def by clarify
  (meson  $\langle$ conflicting-bj-clss S  $\neq$  conflicting-bj-clss T $\rangle$ 
     $\langle$ conflicting-bj-clss S  $\subseteq$  conflicting-bj-clss T $\rangle$ 
    diff-less-mono2 le-less-trans not-le psubsetI)
qed

```

We have to assume the following:

- *inv* S: the invariant holds in the initial state.
- A is a (finite *finite* A) superset of the literals in the trail *atm-of ' lits-of-l* (trail S) \subseteq *atms-of-ms* A and in the clauses *atms-of-mm* (clauses_{NOT} S) \subseteq *atms-of-ms* A. This can 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**

$$\mu_{CDCL} A T \equiv ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)))$$

$$- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T),$$

$$\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T, \text{card} (\text{set-mset} (\text{clauses}_{NOT} T)))$$

lemma *cdcl_{NOT}-decreasing-measure*:

assumes

cdcl_{NOT} S T **and**

inv: *inv* S **and**

atm-clss: *atms-of-mm* (clauses_{NOT} S) \subseteq *atms-of-ms* A **and**

atm-lits: *atm-of ' lits-of-l* (trail S) \subseteq *atms-of-ms* A **and**

n-d: *no-dup* (trail S) **and**

fin-A: *finite* A

shows ($\mu_{CDCL} A T, \mu_{CDCL} A S$)

\in *less-than* $\langle *lex* \rangle$ (*less-than* $\langle *lex* \rangle$ *less-than*)

using *assms*(1)

proof *induction*

case (*c-dpll-bj* T)

from *dpll-bj-trail-mes-decreasing-prop*[OF *this*(1) *inv atm-clss atm-lits n-d fin-A*]

```

show ?case unfolding  $\mu_{CDCL-def}$ 
  by (meson in-lex-prod less-than-iff)
next
case (c-learn  $T$ ) note learn = this(1)
then have  $S$ : trail  $S$  = trail  $T$ 
  using inv atm-clss atm-lits n-d fin- $A$ 
  by (elim learnNOT $E$ ) auto
show ?case
  using learn- $\mu_L$ -decrease[ $OF$  learn  $n-d$ , of atms-of-ms  $A$ ] atm-clss atm-lits fin- $A$   $n-d$ 
  unfolding  $S$   $\mu_{CDCL-def}$  by auto
next
case (c-forgetNOT  $T$ ) note forgetNOT = this(1)
have trail  $S$  = trail  $T$  using forgetNOT by induction auto
then show ?case
  using forget- $\mu_L$ -decrease[ $OF$  forgetNOT] unfolding  $\mu_{CDCL-def}$  by auto
qed

```

lemma wf-cdcl_{NOT}-restricted-learning:

```

assumes finite  $A$ 
shows wf {( $T$ ,  $S$ ).
  (atms-of-mm (clausesNOT  $S$ )  $\subseteq$  atms-of-ms  $A$   $\wedge$  atm-of ' lits-of-l (trail  $S$ )  $\subseteq$  atms-of-ms  $A$ 
 $\wedge$  no-dup (trail  $S$ )
 $\wedge$  inv  $S$ )
 $\wedge$  cdclNOT  $S$   $T$  }
by (rule wf-wf-if-measure'[of less-than <*lex*> (less-than <*lex*> less-than)])
  (auto intro: cdclNOT-decreasing-measure[ $OF$  - - - - assms])

```

definition $\mu_C' :: 'v$ literal multiset set $\Rightarrow 'st \Rightarrow nat$ **where**

$\mu_C' A T \equiv \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$

definition $\mu_{CDCL}' :: 'v$ literal multiset set $\Rightarrow 'st \Rightarrow nat$ **where**

$\mu_{CDCL}' A T \equiv$
 $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A T) * (1 + 3^{\text{card} (\text{atms-of-ms } A)}) * 2$
 $+ \text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T * 2$
 $+ \text{card} (\text{set-mset} (\text{clauses}_{NOT} T))$

lemma cdcl_{NOT}-decreasing-measure':

```

assumes
  cdclNOT  $S$   $T$  and
  inv: inv  $S$  and
  atms-clss: atms-of-mm (clausesNOT  $S$ )  $\subseteq$  atms-of-ms  $A$  and
  atms-trail: atm-of ' lits-of-l (trail  $S$ )  $\subseteq$  atms-of-ms  $A$  and
  n-d: no-dup (trail  $S$ ) and
  fin-A: finite  $A$ 
shows  $\mu_{CDCL}' A T < \mu_{CDCL}' A S$ 
using assms(1)

```

proof (induction rule: cdcl_{NOT}-learn-all-induct)

```

case (dpll-bj  $T$ )
then have  $(2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A T$ 
 $< (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A S$ 
using dpll-bj-trail-mes-decreasing-prop fin- $A$  inv n-d atms-clss atms-trail
unfolding  $\mu_C'$ -def by blast
then have  $XX: ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A T) + 1$ 
 $\leq (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A S$ 

```



```

  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)) -  $\mu_C'$  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)) -  $\mu_C'$  A S)
  * (1 + 3 ^ card (atms-of-ms A))
unfolding Nat.add-mult-distrib
by presburger
moreover
  have cl-T-S: clausesNOT T = clausesNOT S
  using dpll-bj.hyps inv dpll-bj-clauses by auto
  have conflicting-bj-clss-yet (card (atms-of-ms A)) S < 1 + 3 ^ card (atms-of-ms A)
  by simp
ultimately have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) -  $\mu_C'$  A T)
  * (1 + 3 ^ card (atms-of-ms A)) + conflicting-bj-clss-yet (card (atms-of-ms A)) T
< ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) -  $\mu_C'$  A S) * (1 + 3 ^ card (atms-of-ms
A))
  by linarith
then have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) -  $\mu_C'$  A T)
  * (1 + 3 ^ card (atms-of-ms A))
  + conflicting-bj-clss-yet (card (atms-of-ms A)) T
< ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) -  $\mu_C'$  A S)
  * (1 + 3 ^ card (atms-of-ms A))
  + conflicting-bj-clss-yet (card (atms-of-ms A)) S
  by linarith
then have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) -  $\mu_C'$  A T)
  * (1 + 3 ^ card (atms-of-ms A)) * 2
  + conflicting-bj-clss-yet (card (atms-of-ms A)) T * 2
< ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) -  $\mu_C'$  A S)
  * (1 + 3 ^ card (atms-of-ms A)) * 2
  + conflicting-bj-clss-yet (card (atms-of-ms A)) S * 2
  by linarith
then show ?case unfolding  $\mu_{CDCL}'$ -def cl-T-S by presburger
next
case (learn C F' K F C' L T) note clss-S-C = this(1) and atms-C = this(2) and dist = this(3)
  and tauto = this(4) and learn-restr = this(5) and tr-S = this(6) and C' = this(7) and
  F-C = this(8) and C-new = this(9) and T = this(10)
have insert (mset-cls C) (conflicting-bj-clss S) ⊆ simple-clss (atms-of-ms A)
proof -
  have mset-cls C ∈ simple-clss (atms-of-ms A)
  using C'
  by (metis (no-types, hide-lams) Un-subset-iff simple-clss-mono
    contra-subsetD dist distinct-mset-not-tautology-implies-in-simple-clss
    dual-order.trans atms-C atms-clss atms-trail tauto)
  moreover have conflicting-bj-clss S ⊆ simple-clss (atms-of-ms A)
  unfolding conflicting-bj-clss-def
  proof
    fix x :: 'v literal multiset
    assume x ∈ {C + {#L#} | C L. C + {#L#} ∈ # clausesNOT S
      ∧ distinct-mset (C + {#L#}) ∧ ¬ tautology (C + {#L#})
      ∧ (∃ F' K F. trail S = F' @ Marked K () # F ∧ F ⊢as CNot C)}
    then have ∃ m l. x = m + {#l#} ∧ m + {#l#} ∈ # clausesNOT S
      ∧ distinct-mset (m + {#l#}) ∧ ¬ tautology (m + {#l#})
      ∧ (∃ msa l msa. trail S = msa @ Marked l () # msa ∧ msa ⊢as CNot m)
    by blast
  end
end

```

```

    then show  $x \in \text{simple-clss} (\text{atms-of-ms } A)$ 
    by (meson atms-clss atms-of-atms-of-ms-mono atms-of-ms-finite simple-clss-mono
        distinct-mset-not-tautology-implies-in-simple-clss fin-A finite-subset
        set-rev-mp)
    qed
    ultimately show ?thesis
    by auto
    qed
  then have  $\text{card} (\text{insert} (\text{mset-cls } C) (\text{conflicting-bj-clss } S)) \leq 3 \wedge (\text{card} (\text{atms-of-ms } A))$ 
  by (meson Nat.le-trans atms-of-ms-finite simple-clss-card simple-clss-finite
      card-mono fin-A)
  moreover have [simp]:  $\text{card} (\text{insert} (\text{mset-cls } C) (\text{conflicting-bj-clss } S))$ 
    =  $\text{Suc} (\text{card} ((\text{conflicting-bj-clss } S)))$ 
  by (metis (no-types) C' C-new card-insert-if conflicting-bj-clss-incl-clauses contra-subsetD
      finite-conflicting-bj-clss)
  moreover have [simp]:  $\text{conflicting-bj-clss} (\text{add-cl}_{\text{NOT}} C S) = \text{conflicting-bj-clss } S \cup \{\text{mset-cls } C\}$ 
    using dist tauto F-C n-d by (subst conflicting-bj-clss-add-cl_{NOT})
    (force simp add: ac-simps C' tr-S)
  ultimately have [simp]:  $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) S$ 
    =  $\text{Suc} (\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) (\text{add-cl}_{\text{NOT}} C S))$ 
    by simp
  have 1:  $\text{clauses}_{\text{NOT}} T = \text{clauses}_{\text{NOT}} (\text{add-cl}_{\text{NOT}} C S)$  using T by auto
  have 2:  $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T$ 
    =  $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) (\text{add-cl}_{\text{NOT}} C S)$ 
    using T unfolding conflicting-bj-clss-def by auto
  have 3:  $\mu_C' A T = \mu_C' A (\text{add-cl}_{\text{NOT}} C S)$ 
    using T unfolding \mu_C'-def by auto
  have  $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A (\text{add-cl}_{\text{NOT}} C S))$ 
    *  $(1 + 3 \wedge \text{card} (\text{atms-of-ms } A)) * 2$ 
    =  $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A S)$ 
    *  $(1 + 3 \wedge \text{card} (\text{atms-of-ms } A)) * 2$ 
    using n-d unfolding \mu_C'-def by auto
  moreover
    have  $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) (\text{add-cl}_{\text{NOT}} C S)$ 
      * 2
      +  $\text{card} (\text{set-mset} (\text{clauses}_{\text{NOT}} (\text{add-cl}_{\text{NOT}} C S)))$ 
      <  $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) S * 2$ 
      +  $\text{card} (\text{set-mset} (\text{clauses}_{\text{NOT}} S))$ 
    by (simp add: C' C-new n-d)
  ultimately show ?case unfolding  $\mu_{\text{CDCL}}'-\text{def}$  1 2 3 by presburger
next
case (forgetNOT C T) note T = this(4)
have [simp]:  $\mu_C' A (\text{remove-cl}_{\text{NOT}} C S) = \mu_C' A S$ 
  unfolding  $\mu_C'-\text{def}$  by auto
have forgetNOT S T
  apply (rule forgetNOT.intros) using forgetNOT by auto
then have  $\text{conflicting-bj-clss } T = \text{conflicting-bj-clss } S$ 
  using do-not-forget-before-backtrack-rule-clause-learned-clause-untouched by blast
moreover have  $\text{card} (\text{set-mset} (\text{clauses}_{\text{NOT}} T)) < \text{card} (\text{set-mset} (\text{clauses}_{\text{NOT}} S))$ 
  by (metis T card-Diff1-less clauses-remove-cl_{NOT} finite-set-mset forget_{NOT}.hyps(2)
      in-clss-mset-clss order-refl set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
ultimately show ?case unfolding  $\mu_{\text{CDCL}}'-\text{def}$ 
  by (metis (no-types) T \mu_C' A (remove-cl_{NOT} C S) = \mu_C' A S add-le-cancel-left
       $\mu_C'-\text{def not-le state-eq}_{\text{NOT}}-\text{trail}$ )
qed

```

lemma *cdcl_{NOT}-clauses-bound*:

assumes

cdcl_{NOT} S T **and**

inv S **and**

atms-of-mm (clauses_{NOT} S) ⊆ A **and**

atm-of ‘(lits-of-l (trail S)) ⊆ A **and**

n-d: no-dup (trail S) **and**

fin-A[simp]: finite A

shows *set-mset (clauses_{NOT} T) ⊆ set-mset (clauses_{NOT} S) ∪ simple-clss A*

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 (mset-cl_S C) ⊆ A*

using *atms-C atms-clss-S atms-trail-S* **by** *fast*

then have *simple-clss (atms-of (mset-cl_S C)) ⊆ simple-clss A*

by (*simp add: simple-clss-mono*)

then have *mset-cl_S C ∈ simple-clss A*

using *finite dist tauto*

by (*auto dest: distinct-mset-not-tautology-implies-in-simple-clss*)

then show *?case* **using** *T n-d* **by** *auto*

qed

lemma *rtrancp-cdcl_{NOT}-clauses-bound*:

assumes

*cdcl_{NOT}** S T* **and**

inv S **and**

atms-of-mm (clauses_{NOT} S) ⊆ A **and**

atm-of ‘(lits-of-l (trail S)) ⊆ A **and**

n-d: no-dup (trail S) **and**

finite: finite A

shows *set-mset (clauses_{NOT} T) ⊆ set-mset (clauses_{NOT} S) ∪ simple-clss A*

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-S = this(7)*

have *inv T*

using *rtrancp-cdcl_{NOT}-inv st inv* **by** *blast*

moreover have *atms-of-mm (clauses_{NOT} T) ⊆ A* **and** *atm-of ‘ lits-of-l (trail T) ⊆ A*

using *rtrancp-cdcl_{NOT}-trail-clauses-bound[OF st] inv atms-clss-S atms-trail-S n-d* **by** *auto*

moreover have *no-dup (trail T)*

using *rtrancp-cdcl_{NOT}-no-dup[OF st ⟨inv S⟩ n-d]* **by** *simp*

ultimately have *set-mset (clauses_{NOT} U) ⊆ set-mset (clauses_{NOT} T) ∪ 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 *rtranclp-cdcl_{NOT}-card-clauses-bound*:

assumes

*cdcl_{NOT}** S T and*

inv S and

atms-of-mm (clauses_{NOT} S) \subseteq A and

atm-of '(lits-of-l (trail S)) \subseteq A and

n-d: no-dup (trail S) and

finite: finite A

shows *card (set-mset (clauses_{NOT} T)) \leq card (set-mset (clauses_{NOT} S)) + 3 \wedge (card A)*

using *rtranclp-cdcl_{NOT}-clauses-bound[OF assms] finite by (meson Nat.le-trans*

simple-clss-card simple-clss-finite card-Un-le card-mono finite-UnI

finite-set-mset nat-add-left-cancel-le)

lemma *rtranclp-cdcl_{NOT}-card-clauses-bound'*:

assumes

*cdcl_{NOT}** S T and*

inv S and

atms-of-mm (clauses_{NOT} S) \subseteq A and

atm-of '(lits-of-l (trail S)) \subseteq A and

n-d: no-dup (trail S) and

finite: finite A

shows *card {C | C. C \in # clauses_{NOT} T \wedge (tautology C \vee \neg distinct-mset C)}*

\leq card {C | C. C \in # clauses_{NOT} S \wedge (tautology C \vee \neg distinct-mset C)} + 3 \wedge (card A)

(is card ?T \leq card ?S + -)

using *rtranclp-cdcl_{NOT}-clauses-bound[OF assms] finite*

proof –

have *?T \subseteq ?S \cup simple-clss A*

using *rtranclp-cdcl_{NOT}-clauses-bound[OF assms] by force*

then have *card ?T \leq card (?S \cup simple-clss A)*

using *finite by (simp add: assms(5) simple-clss-finite card-mono)*

then show *?thesis*

by *(meson le-trans simple-clss-card card-Un-le local.finite nat-add-left-cancel-le)*

qed

lemma *rtranclp-cdcl_{NOT}-card-simple-clauses-bound*:

assumes

*cdcl_{NOT}** S T and*

inv S and

atms-of-mm (clauses_{NOT} S) \subseteq A and

atm-of '(lits-of-l (trail S)) \subseteq A and

n-d: no-dup (trail S) and

finite: finite A

shows *card (set-mset (clauses_{NOT} T))*

\leq card {C. C \in # clauses_{NOT} S \wedge (tautology C \vee \neg distinct-mset C)} + 3 \wedge (card A)

(is card ?T \leq card ?S + -)

using *rtranclp-cdcl_{NOT}-clauses-bound[OF assms] finite*

proof –

have *$\bigwedge x. x \in$ # clauses_{NOT} T \implies \neg tautology x \implies distinct-mset x \implies x \in simple-clss A*

using *rtranclp-cdcl_{NOT}-clauses-bound[OF assms] by (metis (no-types, hide-lams) Un-iff assms(3)*

atms-of-atms-of-ms-mono simple-clss-mono contra-subsetD

distinct-mset-not-tautology-implies-in-simple-clss subset-trans)

then have *set-mset (clauses_{NOT} T) \subseteq ?S \cup simple-clss A*

using *rtranclp-cdcl_{NOT}-clauses-bound[OF assms] by auto*

then have $\text{card}(\text{set-mset } (\text{clauses}_{\text{NOT}} T)) \leq \text{card } (?S \cup \text{simple-clss } A)$
using *finite* **by** (*simp add: assms(5) simple-clss-finite card-mono*)
then show *?thesis*
by (*meson le-trans simple-clss-card card-Un-le local.finite nat-add-left-cancel-le*)
qed

definition $\mu_{CDCL}'\text{-bound} :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_{CDCL}'\text{-bound } A \ S =$
 $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $+ 2 * 3 \wedge (\text{card } (\text{atms-of-ms } A))$
 $+ \text{card } \{C. C \in \# \text{ clauses}_{\text{NOT}} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } (\text{atms-of-ms } A))$

lemma $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{\text{NOT}}[\text{simp}]$:

$\mu_{CDCL}'\text{-bound } A \ (\text{reduce-trail-to}_{\text{NOT}} M \ S) = \mu_{CDCL}'\text{-bound } A \ S$

unfolding $\mu_{CDCL}'\text{-bound-def}$ **by** *auto*

lemma $\text{rtrancpl-cdcl}_{\text{NOT}}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{\text{NOT}}$:

assumes

$\text{cdcl}_{\text{NOT}}^{**} S \ T$ **and**

$\text{inv } S$ **and**

$\text{atms-of-mm } (\text{clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } (\text{atms-of-ms } A)$ **and**

$U: U \sim \text{reduce-trail-to}_{\text{NOT}} M \ T$

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 $\text{card } (\text{set-mset } (\text{clauses}_{\text{NOT}} U))$

$\leq \text{card } \{C. C \in \# \text{ clauses}_{\text{NOT}} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge \text{card } (\text{atms-of-ms } A)$

using $\text{rtrancpl-cdcl}_{\text{NOT}}\text{-card-simple-clauses-bound}[OF \text{ assms}(1-6)] \ U$ **by** *auto*

ultimately show *?thesis*

unfolding $\mu_{CDCL}'\text{-def}$ $\mu_{CDCL}'\text{-bound-def}$ **by** *linarith*

qed

lemma $\text{rtrancpl-cdcl}_{\text{NOT}}\text{-}\mu_{CDCL}'\text{-bound}$:

assumes

$\text{cdcl}_{\text{NOT}}^{**} S \ T$ **and**

$\text{inv } S$ **and**

$\text{atms-of-mm } (\text{clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } (\text{atms-of-ms } A)$

shows $\mu_{CDCL}' A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$

proof –

have $\mu_{CDCL}' A \ (\text{reduce-trail-to}_{\text{NOT}} (\text{trail } T) \ T) = \mu_{CDCL}' A \ T$

unfolding $\mu_{CDCL}'\text{-def}$ $\mu_C'\text{-def}$ *conflicting-bj-clss-def* **by** *auto*
then show *?thesis* **using** *rtrancpl-cdcl_{NOT}- $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}$ [OF assms, of - trail T]*
state-eq_{NOT}-ref **by** *fastforce*
qed

lemma *rtrancpl- $\mu_{CDCL}'\text{-bound-decreasing}$:*

assumes
*cdcl_{NOT}** S T and*
inv S and
atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A and
atm-of '(lits-of-l (trail S)) \subseteq atms-of-ms A and
n-d: no-dup (trail S) and
finite[simp]: finite (atms-of-ms A)
shows $\mu_{CDCL}'\text{-bound A T} \leq \mu_{CDCL}'\text{-bound A S}$
proof –
have $\{C. C \in \# \text{ clauses}_{NOT} T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$
 $\subseteq \{C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$ **(is ?T \subseteq ?S)**
proof (*rule Set.subsetI*)
fix *C* **assume** $C \in ?T$
then have *C-T: C $\in \#$ clauses_{NOT} T and t-d: tautology C \vee \neg distinct-mset C*
by *auto*
then have *C \notin simple-clss (atms-of-ms A)*
by (*auto dest: simple-clssE*)
then show $C \in ?S$
using *C-T rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] t-d by force*
qed
then have $\text{card } \{C. C \in \# \text{ clauses}_{NOT} T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} \leq$
 $\text{card } \{C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$
by (*simp add: card-mono*)
then show *?thesis*
unfolding $\mu_{CDCL}'\text{-bound-def}$ **by** *auto*
qed

end — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learn*

15.7 CDCL with restarts

15.7.1 Definition

locale *restart-ops* =
fixes
cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool and
restart :: 'st \Rightarrow 'st \Rightarrow bool
begin
inductive *cdcl_{NOT}-raw-restart :: 'st \Rightarrow 'st \Rightarrow bool where*
cdcl_{NOT} S T \Longrightarrow cdcl_{NOT}-raw-restart S T |
restart S T \Longrightarrow cdcl_{NOT}-raw-restart S T

end

locale *conflict-driven-clause-learning-with-restarts* =
conflict-driven-clause-learning mset-cls union-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
inv backjump-conds propagate-conds learn-cond forget-cond
for

```

mset-cls:: 'cls ⇒ 'v clause and
union-cls :: 'cls ⇒ 'cls ⇒ 'cls and
insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls and
remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls and
mset-clss:: 'clss ⇒ 'v clauses and
union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
in-clss :: 'cls ⇒ 'clss ⇒ bool and
insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and
trail :: 'st ⇒ ('v, unit, unit) marked-lits and
raw-clauses :: 'st ⇒ 'clss and
prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
tl-trail :: 'st ⇒ 'st and
add-clNOT :: 'cls ⇒ 'st ⇒ 'st and
remove-clNOT :: 'cls ⇒ 'st ⇒ 'st and
inv :: 'st ⇒ bool and
backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
learn-cond forget-cond :: 'cls ⇒ 'st ⇒ bool
begin

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
  cdclNOT S T ⟷ restart-ops.cdclNOT-raw-restart cdclNOT (λ-. False) S T
  (is ?C S T ⟷ ?R S T)
proof
  fix S T
  assume ?C S T
  then show ?R S T by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
next
  fix S T
  assume ?R S T
  then show ?C S T
    apply (cases rule: restart-ops.cdclNOT-raw-restart.cases)
    using ⟨?R S T⟩ by fast+
qed

lemma cdclNOT-cdclNOT-raw-restart:
  cdclNOT S T ⟹ restart-ops.cdclNOT-raw-restart cdclNOT restart S T
  by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
end

```

15.7.2 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 $(1::'a) \leq f$ for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...
- a measure μ : it should decrease under the assumptions $bound-inv$, whenever a $cdcl_{NOT}$ or a $restart$ is done. A parameter is given to μ : for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.

- we also assume that the measure decrease after any $cdcl_{NOT}$ step.
- an invariant on the states $cdcl_{NOT}$ -inv that also holds after restarts.
- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ -bound taking the same parameter as μ and the initial state of the considered $cdcl_{NOT}$ chain.

```

locale  $cdcl_{NOT}$ -increasing-restarts-ops =
  restart-ops  $cdcl_{NOT}$  restart for
    restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
     $cdcl_{NOT}$  :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool +
fixes
  f :: nat  $\Rightarrow$  nat and
  bound-inv :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  bool and
   $\mu$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat and
   $cdcl_{NOT}$ -inv :: 'st  $\Rightarrow$  bool and
   $\mu$ -bound :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat
assumes
  f: unbounded f and
  f-ge-1:  $\bigwedge n. n \geq 1 \Rightarrow f\ n \neq 0$  and
  bound-inv:  $\bigwedge A\ S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$  bound-inv A S  $\Rightarrow$   $cdcl_{NOT}$  S T  $\Rightarrow$  bound-inv A T and
   $cdcl_{NOT}$ -measure:  $\bigwedge A\ S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$  bound-inv A S  $\Rightarrow$   $cdcl_{NOT}$  S T  $\Rightarrow$   $\mu\ A\ T < \mu$ 
A S and
  measure-bound2:  $\bigwedge A\ T\ U. cdcl_{NOT}$ -inv T  $\Rightarrow$  bound-inv A T  $\Rightarrow$   $cdcl_{NOT}^{**}$  T U
     $\Rightarrow \mu\ A\ U \leq \mu$ -bound A T and
  measure-bound4:  $\bigwedge A\ T\ U. cdcl_{NOT}$ -inv T  $\Rightarrow$  bound-inv A T  $\Rightarrow$   $cdcl_{NOT}^{**}$  T U
     $\Rightarrow \mu$ -bound A U  $\leq \mu$ -bound A T and
   $cdcl_{NOT}$ -restart-inv:  $\bigwedge A\ U\ V. cdcl_{NOT}$ -inv U  $\Rightarrow$  restart U V  $\Rightarrow$  bound-inv A U  $\Rightarrow$  bound-inv
A V
and
  exists-bound:  $\bigwedge R\ S. cdcl_{NOT}$ -inv R  $\Rightarrow$  restart R S  $\Rightarrow$   $\exists A. bound$ -inv A S and
   $cdcl_{NOT}$ -inv:  $\bigwedge S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$   $cdcl_{NOT}$  S T  $\Rightarrow$   $cdcl_{NOT}$ -inv T and
   $cdcl_{NOT}$ -inv-restart:  $\bigwedge S\ T. cdcl_{NOT}$ -inv S  $\Rightarrow$  restart S T  $\Rightarrow$   $cdcl_{NOT}$ -inv T
begin

```

lemma $cdcl_{NOT}$ - $cdcl_{NOT}$ -inv:

```

assumes
  ( $cdcl_{NOT} \sim n$ ) S T and
   $cdcl_{NOT}$ -inv S
shows  $cdcl_{NOT}$ -inv T
using assms by (induction n arbitrary: T) (auto intro: bound-inv  $cdcl_{NOT}$ -inv)

```

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

```

assumes
  ( $cdcl_{NOT} \sim n$ ) S T and
   $cdcl_{NOT}$ -inv S
  bound-inv A S
shows bound-inv A T
using assms by (induction n arbitrary: T) (auto intro: bound-inv  $cdcl_{NOT}$ - $cdcl_{NOT}$ -inv)

```

lemma $rtrancp$ - $cdcl_{NOT}$ - $cdcl_{NOT}$ -inv:

```

assumes
   $cdcl_{NOT}^{**}$  S T and
   $cdcl_{NOT}$ -inv S

```


shows $cdcl_{NOT-inv} T$
using *assms* **by** *induction* (*auto intro: cdcl_{NOT-inv}*)

lemma *rtranclp-cdcl_{NOT-bound-inv}*:
assumes
 $cdcl_{NOT}^{**} S T$ **and**
 $bound-inv A S$ **and**
 $cdcl_{NOT-inv} S$
shows $bound-inv A T$
using *assms* **by** *induction* (*auto intro: bound-inv rtranclp-cdcl_{NOT}-cdcl_{NOT-inv}*)

lemma *cdcl_{NOT-comp-n-le}*:
assumes
 $(cdcl_{NOT} \sim (Suc\ n)) S T$ **and**
 $bound-inv A S$
 $cdcl_{NOT-inv} S$
shows $\mu A T < \mu A S - n$
using *assms*
proof (*induction n arbitrary: T*)
case 0
then show ?*case* **using** *cdcl_{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*s)
moreover have $cdcl_{NOT-inv} T$

using $c\text{-inv}$ $r\text{trancp-cdcl}_{NOT}\text{-cdcl}_{NOT}\text{-inv}$ st by *blast*
 ultimately have $\mu A U < \mu A T$ using $cdcl_{NOT}\text{-measure}[OF - - cdcl_{NOT}]$ by *auto*
 then show $?case$ using IH by *linarith*
 qed

lemma $cdcl_{NOT}\text{-comp-bounded}$:

assumes

$bound\text{-inv } A S$ and $cdcl_{NOT}\text{-inv } S$ and $m \geq 1 + \mu A S$

shows $\neg(cdcl_{NOT} \rightsquigarrow^m) S T$

using *assms* $cdcl_{NOT}\text{-comp-n-le}[of m-1 S T A]$ by *fastforce*

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

inductive $cdcl_{NOT}\text{-restart}$ where

restart-step: $(cdcl_{NOT} \rightsquigarrow^m) S T \implies m \geq f n \implies \text{restart } T U$

$\implies cdcl_{NOT}\text{-restart } (S, n) (U, Suc n) \mid$

restart-full: $full1\ cdcl_{NOT} S T \implies cdcl_{NOT}\text{-restart } (S, n) (T, Suc n)$

lemmas $cdcl_{NOT}\text{-with-restart-induct} = cdcl_{NOT}\text{-restart.induct}[split\text{-format}(complete),$
 $OF\ cdcl_{NOT}\text{-increasing-restarts-ops-axioms}]$

lemma $cdcl_{NOT}\text{-restart-cdcl}_{NOT}\text{-raw-restart}$:

$cdcl_{NOT}\text{-restart } S T \implies cdcl_{NOT}\text{-raw-restart}^{**} (fst S) (fst T)$

proof (*induction rule*: $cdcl_{NOT}\text{-restart.induct}$)

case (*restart-step* $m S T n U$)

then have $cdcl_{NOT}^{**} S T$ by (*meson relpowp-imp-rtrancp*)

then have $cdcl_{NOT}\text{-raw-restart}^{**} S T$ using $cdcl_{NOT}\text{-raw-restart.intros}(1)$

$r\text{trancp-mono}[of\ cdcl_{NOT}\ cdcl_{NOT}\text{-raw-restart}]$ by *blast*

moreover have $cdcl_{NOT}\text{-raw-restart } T U$

using $\langle \text{restart } T U \rangle cdcl_{NOT}\text{-raw-restart.intros}(2)$ by *blast*

ultimately show $?case$ by *auto*

next

case (*restart-full* $S T$)

then have $cdcl_{NOT}^{**} S T$ unfolding *full1-def* by *auto*

then show $?case$ using $cdcl_{NOT}\text{-raw-restart.intros}(1)$

$r\text{trancp-mono}[of\ cdcl_{NOT}\ cdcl_{NOT}\text{-raw-restart}]$ by *auto*

qed

lemma $cdcl_{NOT}\text{-with-restart-bound-inv}$:

assumes

$cdcl_{NOT}\text{-restart } S T$ and

$bound\text{-inv } A (fst S)$ and

$cdcl_{NOT}\text{-inv } (fst S)$

shows $bound\text{-inv } A (fst T)$

using *assms* **apply** (*induction rule*: $cdcl_{NOT}\text{-restart.induct}$)

prefer 2 **apply** (*metis* $r\text{trancp-unfold}$ *fstI* *full1-def* $r\text{trancp-cdcl}_{NOT}\text{-bound-inv}$)

by (*metis* $cdcl_{NOT}\text{-bound-inv}$ $cdcl_{NOT}\text{-cdcl}_{NOT}\text{-inv}$ $cdcl_{NOT}\text{-restart-inv}$ *fst-conv*)

lemma $cdcl_{NOT}\text{-with-restart-cdcl}_{NOT}\text{-inv}$:

assumes

$cdcl_{NOT}\text{-restart } S T$ and

$cdcl_{NOT}\text{-inv } (fst S)$

shows $cdcl_{NOT}\text{-inv } (fst T)$

using *assms* **apply** *induction*

apply (*metis* $cdcl_{NOT}\text{-cdcl}_{NOT}\text{-inv}$ $cdcl_{NOT}\text{-inv-restart}$ *fst-conv*)

apply (*metis fstI full-def full-unfold rtrancp-cdcl_{NOT}-cdcl_{NOT}-inv*)
done

lemma *rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*:

assumes
*cdcl_{NOT}-restart** S T and*
cdcl_{NOT}-inv (fst S)
shows *cdcl_{NOT}-inv (fst T)*
using *assms* **by** *induction (auto intro: cdcl_{NOT}-with-restart-cdcl_{NOT}-inv)*

lemma *rtrancp-cdcl_{NOT}-with-restart-bound-inv*:

assumes
*cdcl_{NOT}-restart** S T and*
cdcl_{NOT}-inv (fst S) and
bound-inv A (fst S)
shows *bound-inv A (fst T)*
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 rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv* **by** *blast*

lemma *cdcl_{NOT}-with-restart-increasing-number*:

cdcl_{NOT}-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 +
dpll-state mset-cls union-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT}

for

mset-cls:: 'cls \Rightarrow 'v clause and
union-cls :: 'cls \Rightarrow 'cls \Rightarrow 'cls and
insert-cls :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and
remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and
mset-clss:: 'clss \Rightarrow 'v clauses and
union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss and
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool and
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and
remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and
trail :: 'st \Rightarrow ('v, unit, unit) marked-lits and
raw-clauses :: 'st \Rightarrow 'clss and
prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st and
tl-trail :: 'st \Rightarrow 'st and
add-cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st and
remove-cls_{NOT} :: 'cls \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: $\bigwedge A T V n. cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A T$

$\implies \text{cdcl}_{NOT}\text{-restart } (T, n) (V, \text{Suc } n) \implies \mu A V \leq \mu\text{-bound } A T$ **and**
 $\text{cdcl}_{NOT}\text{-raw-restart-}\mu\text{-bound}:$
 $\text{cdcl}_{NOT}\text{-restart } (T, a) (V, b) \implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A T$
 $\implies \mu\text{-bound } A V \leq \mu\text{-bound } A T$

begin

lemma *rtrancp-cdcl_{NOT}-raw-restart- μ -bound:*
 $\text{cdcl}_{NOT}\text{-restart}^{**} (T, a) (V, b) \implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A T$
 $\implies \mu\text{-bound } A V \leq \mu\text{-bound } A T$
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:*
 $\text{cdcl}_{NOT}\text{-restart } (T, a) (V, b) \implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A T$
 $\implies \mu A V \leq \mu\text{-bound } A T$
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:*
 $\text{cdcl}_{NOT}\text{-restart}^{**} (T, a) (V, b) \implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A T$
 $\implies \mu A V \leq \mu\text{-bound } A T$
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:*
 $\text{wf } \{(T, S). \text{cdcl}_{NOT}\text{-restart } S T \wedge \text{cdcl}_{NOT}\text{-inv } (\text{fst } S)\}$ (**is** *wf ?A*)

proof (*rule ccontr*)
assume $\neg ?thesis$
then obtain *g* **where**
 $g: \bigwedge i. \text{cdcl}_{NOT}\text{-restart } (g \ i) (g (\text{Suc } i))$ **and**
 $\text{cdcl}_{NOT}\text{-inv-}g: \bigwedge i. \text{cdcl}_{NOT}\text{-inv } (\text{fst } (g \ i))$
unfolding *wf-iff-no-infinite-down-chain* **by** *fast*

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.commute add.left-commute*
cdcl_{NOT}-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*: $\text{unbounded } (\lambda i. f (\text{snd } (g \ i)))$
using *f* **unfolding** *bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*
not-bounded-nat-exists-larger not-le le-iff-add)

{ fix *i*
have *H*: $\bigwedge T \text{ Ta } m. (\text{cdcl}_{NOT} \rightsquigarrow m) T \text{ Ta} \implies \text{no-step } \text{cdcl}_{NOT} T \implies m = 0$
apply (*case-tac m*) **by** *simp (meson relpowp-E2)*
have $\exists T m. (\text{cdcl}_{NOT} \rightsquigarrow m) (\text{fst } (g \ i)) T \wedge m \geq f (\text{snd } (g \ i))$

```

    using  $g[of\ i]$  apply (cases rule:  $cdcl_{NOT}$ -restart.cases)
      apply auto[]
    using  $g[of\ Suc\ i]$   $f\text{-}ge\text{-}1$  apply (cases rule:  $cdcl_{NOT}$ -restart.cases)
    apply (auto simp add: full1-def full-def dest: H dest: tranclpD)
    using  $H\ Suc\ leI\ leD$  by blast
  } note  $H = this$ 
obtain  $A$  where  $bound\text{-}inv\ A\ (fst\ (g\ 1))$ 
  using  $g[of\ 0]$   $cdcl_{NOT}\text{-}inv\text{-}g[of\ 0]$  apply (cases rule:  $cdcl_{NOT}$ -restart.cases)
  apply (metis One-nat-def cdclNOT-inv exists-bound fst-conv relpowp-imp-rtranclp
    rtranclp-induct)
  using  $H[of\ 1]$  unfolding full1-def by (metis One-nat-def Suc-eq-plus1 diff-is-0-eq' diff-zero
    f-ge-1 fst-conv le-add2 relpowp-E2 snd-conv)
let  $?j = \mu\text{-}bound\ A\ (fst\ (g\ 1)) + 1$ 
obtain  $j$  where
   $j: f\ (snd\ (g\ j)) > ?j$  and  $j > 1$ 
  using unbounded-f-g not-bounded-nat-exists-larger by blast
{
  fix  $i\ j$ 
  have  $cdcl_{NOT}\text{-}with\text{-}restart: j \geq i \implies cdcl_{NOT}\text{-}restart^{**}\ (g\ i)\ (g\ j)$ 
    apply (induction j)
    apply simp
    by (metis g le-Suc-eq rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
} note  $cdcl_{NOT}\text{-}restart = this$ 
have  $cdcl_{NOT}\text{-}inv\ (fst\ (g\ (Suc\ 0)))$ 
  by (simp add: cdclNOT-inv-g)
have  $cdcl_{NOT}\text{-}restart^{**}\ (fst\ (g\ 1),\ snd\ (g\ 1))\ (fst\ (g\ j),\ snd\ (g\ j))$ 
  using  $\langle j > 1 \rangle$  by (simp add: cdclNOT-restart)
have  $\mu\ A\ (fst\ (g\ j)) \leq \mu\text{-}bound\ A\ (fst\ (g\ 1))$ 
  apply (rule rtranclp-cdclNOT-raw-restart-measure-bound)
  using  $\langle cdcl_{NOT}\text{-}restart^{**}\ (fst\ (g\ 1),\ snd\ (g\ 1))\ (fst\ (g\ j),\ snd\ (g\ j)) \rangle$  apply blast
  apply (simp add: cdclNOT-inv-g)
  using  $\langle bound\text{-}inv\ A\ (fst\ (g\ 1)) \rangle$  apply simp
  done
then have  $\mu\ A\ (fst\ (g\ j)) \leq ?j$ 
  by auto
have  $inv: bound\text{-}inv\ A\ (fst\ (g\ j))$ 
  using  $\langle bound\text{-}inv\ A\ (fst\ (g\ 1)) \rangle\ \langle cdcl_{NOT}\text{-}inv\ (fst\ (g\ (Suc\ 0))) \rangle$ 
   $\langle cdcl_{NOT}\text{-}restart^{**}\ (fst\ (g\ 1),\ snd\ (g\ 1))\ (fst\ (g\ j),\ snd\ (g\ j)) \rangle$ 
  rtranclp-cdclNOT-with-restart-bound-inv by auto
obtain  $T\ m$  where
   $cdcl_{NOT}\text{-}m: (cdcl_{NOT} \rightsquigarrow m)\ (fst\ (g\ j))\ T$  and
   $f\text{-}m: f\ (snd\ (g\ j)) \leq m$ 
  using  $H[of\ j]$  by blast
have  $?j < m$ 
  using  $f\text{-}m\ j\ Nat.le\text{-}trans$  by linarith

then show False
  using  $\langle \mu\ A\ (fst\ (g\ j)) \leq \mu\text{-}bound\ A\ (fst\ (g\ 1)) \rangle$ 
   $cdcl_{NOT}\text{-}comp\text{-}bounded[OF\ inv\ cdcl_{NOT}\text{-}inv\text{-}g,\ of\ ]\ cdcl_{NOT}\text{-}inv\text{-}g\ cdcl_{NOT}\text{-}m$ 
   $\langle ?j < m \rangle$  by auto
qed

```

lemma $cdcl_{NOT}\text{-}restart\text{-}steps\text{-}bigger\text{-}than\text{-}bound$:

assumes

$cdcl_{NOT}\text{-}restart\ S\ T$ **and**

$\text{bound-inv } A \text{ (fst } S) \text{ and}$
 $\text{cdcl}_{NOT}\text{-inv (fst } S) \text{ and}$
 $f \text{ (snd } S) > \mu\text{-bound } A \text{ (fst } S)$
shows $\text{full1 cdcl}_{NOT} \text{ (fst } S) \text{ (fst } T)$
using assms
proof ($\text{induction rule: cdcl}_{NOT}\text{-restart.induct}$)
case restart-full
then show $?case$ **by** auto
next
case ($\text{restart-step } m \ S \ T \ n \ U$) **note** $st = \text{this}(1)$ **and** $f = \text{this}(2)$ **and** $\text{bound-inv} = \text{this}(4)$ **and**
 $\text{cdcl}_{NOT}\text{-inv} = \text{this}(5)$ **and** $\mu = \text{this}(6)$
then obtain m' **where** $m: m = \text{Suc } m'$ **by** ($\text{cases } m$) auto
have $\mu \ A \ S - m' = 0$
using $f \text{ bound-inv cdcl}_{NOT}\text{-inv } \mu \ m \ \text{rtrancp-cdcl}_{NOT}\text{-raw-restart-measure-bound}$ **by** fastforce
then have False **using** $\text{cdcl}_{NOT}\text{-comp-n-le[of } m' \ S \ T \ A]$ restart-step **unfolding** m **by** simp
then show $?case$ **by** fast
qed

lemma $\text{rtrancp-cdcl}_{NOT}\text{-with-inv-inv-rtrancp-cdcl}_{NOT}$:
assumes
 $\text{inv: cdcl}_{NOT}\text{-inv } S$ **and**
 $\text{binv: bound-inv } A \ S$
shows $(\lambda S \ T. \text{cdcl}_{NOT} \ S \ T \wedge \text{cdcl}_{NOT}\text{-inv } S \wedge \text{bound-inv } A \ S)^{**} \ S \ T \longleftrightarrow \text{cdcl}_{NOT}^{**} \ S \ T$
 $(\text{is } ?A^{**} \ S \ T \longleftrightarrow ?B^{**} \ S \ T)$
apply (rule iffI)
using $\text{rtrancp-mono[of } ?A \ ?B]$ **apply** blast
apply ($\text{induction rule: rtrancp-induct}$)
using inv binv **apply** simp
by ($\text{metis (mono-tags, lifting) binv inv rtrancp.simps rtrancp-cdcl}_{NOT}\text{-bound-inv}$
 $\text{rtrancp-cdcl}_{NOT}\text{-cdcl}_{NOT}\text{-inv}$)

lemma $\text{no-step-cdcl}_{NOT}\text{-restart-no-step-cdcl}_{NOT}$:
assumes
 $n\text{-s: no-step cdcl}_{NOT}\text{-restart } S$ **and**
 $\text{inv: cdcl}_{NOT}\text{-inv (fst } S) \text{ and}$
 $\text{binv: bound-inv } A \text{ (fst } S)$
shows $\text{no-step cdcl}_{NOT} \text{ (fst } S)$
proof (rule ccontr)
assume $\neg ?thesis$
then obtain T **where** $T: \text{cdcl}_{NOT} \text{ (fst } S) \ T$
by blast
then obtain U **where** $U: \text{full } (\lambda S \ T. \text{cdcl}_{NOT} \ S \ T \wedge \text{cdcl}_{NOT}\text{-inv } S \wedge \text{bound-inv } A \ S) \ T \ U$
using $\text{wf-exists-normal-form-full[OF wf-cdcl}_{NOT}, \text{ of } A \ T]$ **by** auto
moreover have $\text{inv-T: cdcl}_{NOT}\text{-inv } T$
using $\langle \text{cdcl}_{NOT} \text{ (fst } S) \ T \rangle \text{cdcl}_{NOT}\text{-inv inv}$ **by** blast
moreover have $\text{b-inv-T: bound-inv } A \ T$
using $\langle \text{cdcl}_{NOT} \text{ (fst } S) \ T \rangle \text{binv bound-inv inv}$ **by** blast
ultimately have $\text{full cdcl}_{NOT} \ T \ U$
using $\text{rtrancp-cdcl}_{NOT}\text{-with-inv-inv-rtrancp-cdcl}_{NOT} \ \text{rtrancp-cdcl}_{NOT}\text{-bound-inv}$
 $\text{rtrancp-cdcl}_{NOT}\text{-cdcl}_{NOT}\text{-inv}$ **unfolding** full-def **by** blast
then have $\text{full1 cdcl}_{NOT} \text{ (fst } S) \ U$
using $T \ \text{full-fullI}$ **by** metis
then show False **by** ($\text{metis n-s prod.collapse restart-full}$)
qed

end

15.8 Merging backjump and learning

```

locale cdclNOT-merge-bj-learn-ops =
  decide-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT +
  forget-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT forget-cond +
  propagate-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool +
fixes backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  '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-toNOT F (add-clNOT C'' S))
   $\Rightarrow$  C  $\in$  # clausesNOT S
   $\Rightarrow$  trail S  $\models_{as}$  CNot C
   $\Rightarrow$  undefined-lit F L
   $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))
   $\Rightarrow$  clausesNOT S  $\models_{pm}$  C' + {#L#}
   $\Rightarrow$  mset-cls C'' = C' + {#L#}
   $\Rightarrow$  F  $\models_{as}$  CNot C'
   $\Rightarrow$  backjump-l-cond C C' L S T
   $\Rightarrow$  backjump-l S T

```

Avoid (meaningless) simplification:

```

declare reduce-trail-toNOT-length-ne[simp del] Set.Un-iff[simp del] Set.insert-iff[simp del]
inductive-cases backjump-lE: backjump-l S T
thm backjump-lE
declare reduce-trail-toNOT-length-ne[simp] Set.Un-iff[simp] Set.insert-iff[simp]

```

```

inductive cdclNOT-merged-bj-learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
  cdclNOT-merged-bj-learn-decideNOT: decideNOT S S'  $\Longrightarrow$  cdclNOT-merged-bj-learn S S' |

```

$cdcl_{NOT}\text{-merged-bj-learn-propagate}_{NOT}: propagate_{NOT} S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S' \mid$
 $cdcl_{NOT}\text{-merged-bj-learn-backjump-l}: backjump-l S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S' \mid$
 $cdcl_{NOT}\text{-merged-bj-learn-forget}_{NOT}: forget_{NOT} S S' \Rightarrow cdcl_{NOT}\text{-merged-bj-learn} S S'$

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 $cdcl_{NOT}\text{-merge-bj-learn-proxy} =$

$cdcl_{NOT}\text{-merge-bj-learn-ops}$ $mset\text{-cls}$ $union\text{-cls}$ $insert\text{-cls}$ $remove\text{-lit}$
 $mset\text{-clss}$ $union\text{-clss}$ $in\text{-clss}$ $insert\text{-clss}$ $remove\text{-from-clss}$
 $trail$ $raw\text{-clauses}$ $prepend\text{-trail}$ $tl\text{-trail}$ $add\text{-cls}_{NOT}$ $remove\text{-cls}_{NOT}$ $propagate\text{-conds}$
 $forget\text{-cond}$

$\lambda C C' L' S T. backjump-l\text{-cond } C C' L' S T$

$\wedge distinct\text{-mset } (C' + \{\#L'\# \}) \wedge \neg tautology (C' + \{\#L'\# \})$

for

$mset\text{-cls} :: 'cls \Rightarrow 'v \text{ clause}$ **and**
 $union\text{-cls} :: 'cls \Rightarrow 'cls \Rightarrow 'cls$ **and**
 $insert\text{-cls} :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**
 $remove\text{-lit} :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**
 $mset\text{-clss} :: 'clss \Rightarrow 'v \text{ clauses}$ **and**
 $union\text{-clss} :: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
 $in\text{-clss} :: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**
 $insert\text{-clss} :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
 $remove\text{-from-clss} :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
 $trail :: 'st \Rightarrow ('v, unit, unit) \text{ marked-lits}$ **and**
 $raw\text{-clauses} :: 'st \Rightarrow 'clss$ **and**
 $prepend\text{-trail} :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$ **and**
 $tl\text{-trail} :: 'st \Rightarrow 'st$ **and**
 $add\text{-cls}_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
 $remove\text{-cls}_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
 $propagate\text{-conds} :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow bool$ **and**
 $forget\text{-cond} :: 'cls \Rightarrow 'st \Rightarrow bool$ **and**
 $backjump-l\text{-cond} :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow bool +$

fixes

$inv :: 'st \Rightarrow bool$

assumes

$bj\text{-merge-can-jump}$:

$\bigwedge S C F' K F L.$

$inv S$

$\Rightarrow trail S = F' @ Marked K () \# F$

$\Rightarrow C \in \# clauses_{NOT} S$

$\Rightarrow trail S \models_{as} CNot C$

$\Rightarrow undefined\text{-lit } F L$

$\Rightarrow atm\text{-of } L \in atm\text{-of-mm } (clauses_{NOT} S) \cup atm\text{-of } ' (lits\text{-of-l } (F' @ Marked K () \# F))$

$\Rightarrow clauses_{NOT} S \models_{pm} C' + \{\#L'\# \}$

$\Rightarrow F \models_{as} CNot C'$

$\Rightarrow \neg no\text{-step } backjump-l S$ **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-conds* :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool

where

backjump-conds $\equiv \lambda C\ C'\ L'\ S\ T. \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$

Without additional knowledge on *backjump-l-cond*, it is impossible to have the same invariant.

sublocale *dpll-with-backjumping-ops* *mset-cls* *union-cls* *insert-cls* *remove-lit*
mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*
trail *raw-clauses* *prepend-trail* *tl-trail* *add-clss_{NOT}* *remove-clss_{NOT}* *inv*
backjump-conds *propagate-conds*

proof (*unfold-locales*, *goal-cases*)

case 1

{ **fix** *S S'*

assume *bj*: *backjump-l S S'* **and** *no-dup* (*trail S*)

then obtain *F' K F L C' C D* **where**

S': *S' \sim prepend-trail* (*Propagated L* ()) (*reduce-trail-to_{NOT} F* (*add-clss_{NOT} D S*))

and

tr-S: *trail S = F' @ Marked K* () $\#$ *F* **and**

C: *C $\in \#$ clauses_{NOT} S* **and**

tr-S-C: *trail S \models_{as} CNot C* **and**

undef-L: *undefined-lit F L* **and**

atm-L:

atm-of L \in insert (*atm-of K*) (*atms-of-mm* (*clauses_{NOT} S*) \cup *atm-of 'lit-of ' (set F' \cup set F)*))

and

cls-S-C': *clauses_{NOT} S \models_{pm} C' + $\{\#L\#\}$* **and**

F-C': *F \models_{as} CNot C'* **and**

dist: *distinct-mset* (*C' + $\{\#L\#\}$*) **and**

not-tauto: \neg *tautology* (*C' + $\{\#L\#\}$*) **and**

cond: *backjump-l-cond C C' L S S'*

mset-clss D = C' + $\{\#L\#\}$

by (*elim backjump-lE*) *metis*

interpret *backjumping-ops* *mset-cls* *union-cls* *insert-cls* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-clss_{NOT}* *remove-clss_{NOT}*

backjump-conds

by *unfold-locales*

have $\exists T. \text{backjump } S\ T$

apply *rule*

apply (*rule backjump.intros*)

using *tr-S* **apply** *simp*

apply (*rule state-eq_{NOT}-ref*)

using *C* **apply** *simp*

using *tr-S-C* **apply** *simp*

using *undef-L* **apply** *simp*

using *atm-L tr-S* **apply** *simp*

using *cls-S-C'* **apply** *simp*

using *F-C'* **apply** *simp*

using *dist not-tauto cond* **apply** *simp*

done

} **note** *H = this(1)*

then show ?*case* **using** 1 *bj-merge-can-jump* **by** *meson*

qed

end

```

locale cdclNOT-merge-bj-learn-proxy2 =
  cdclNOT-merge-bj-learn-proxy mset-cls union-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
    propagate-conds forget-cond backjump-l-cond inv
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool
begin

sublocale conflict-driven-clause-learning-ops mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds
   $\lambda C \cdot$  distinct-mset (mset-cls C)  $\wedge$   $\neg$ tautology (mset-cls C)
  forget-cond
by unfold-locales
end

locale cdclNOT-merge-bj-learn =
  cdclNOT-merge-bj-learn-proxy2 mset-cls union-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
    propagate-conds forget-cond backjump-l-cond inv
for
  mset-cls:: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and

```

$tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
 $add\text{-}cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
 $remove\text{-}cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
 $backjump\text{-}l\text{-}cond :: 'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **and**
 $propagate\text{-}conds :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow bool$ **and**
 $forget\text{-}cond :: 'cls \Rightarrow 'st \Rightarrow bool$ **and**
 $inv :: 'st \Rightarrow bool +$
assumes
 $dpll\text{-}bj\text{-}inv: \bigwedge S\ T. \ dpll\text{-}bj\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$ **and**
 $learn\text{-}inv: \bigwedge S\ T. \ learn\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
begin

sublocale
 $conflict\text{-}driven\text{-}clause\text{-}learning\ mset\text{-}cls\ union\text{-}cls\ insert\text{-}cls\ remove\text{-}lit$
 $mset\text{-}clss\ union\text{-}clss\ in\text{-}clss\ insert\text{-}clss\ remove\text{-}from\text{-}clss$
 $trail\ raw\text{-}clauses\ prepend\text{-}trail\ tl\text{-}trail\ add\text{-}cls_{NOT}\ remove\text{-}cls_{NOT}$
 $inv\ backjump\text{-}conds\ propagate\text{-}conds$
 $\lambda C\ -. \ distinct\text{-}mset\ (mset\text{-}cls\ C) \wedge \neg tautology\ (mset\text{-}cls\ C)$
 $forget\text{-}cond$
apply $unfold\text{-}locales$
using $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}forget_{NOT}\ cdcl\text{-}merged\text{-}inv\ learn\text{-}inv$
by $(auto\ simp\ add: cdcl_{NOT}.simps\ dpll\text{-}bj\text{-}inv)$

lemma $backjump\text{-}l\text{-}learn\text{-}backjump$:
assumes $bt: backjump\text{-}l\ S\ T$ **and** $inv: inv\ S$ **and** $n\text{-}d: no\text{-}dup\ (trail\ S)$
shows $\exists C'\ L\ D. \ learn\ S\ (add\text{-}cls_{NOT}\ D\ S)$
 $\wedge mset\text{-}cls\ D = (C' + \{\#L\# \})$
 $\wedge backjump\ (add\text{-}cls_{NOT}\ D\ S)\ T$
 $\wedge atms\text{-}of\ (C' + \{\#L\# \}) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S))$
proof –
obtain $C\ F'\ K\ F\ L\ l\ C'\ D$ **where**
 $tr\text{-}S: trail\ S = F' @ \text{Marked}\ K\ ()\ \# \ F$ **and**
 $T: T \sim prepend\text{-}trail\ (\text{Propagated}\ L\ l)\ (\text{reduce}\text{-}trail\text{-}to_{NOT}\ F\ (add\text{-}cls_{NOT}\ D\ S))$ **and**
 $C\text{-}cls\text{-}S: C \in \# \ clauses_{NOT}\ S$ **and**
 $tr\text{-}S\text{-}CNot\text{-}C: trail\ S \models_{as} CNot\ C$ **and**
 $undef: undefined\text{-}lit\ F\ L$ **and**
 $atm\text{-}L: atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S))$ **and**
 $clss\text{-}C: clauses_{NOT}\ S \models_{pm} mset\text{-}cls\ D$ **and**
 $D: mset\text{-}cls\ D = C' + \{\#L\# \}$
 $F \models_{as} CNot\ C'$ **and**
 $distinct: distinct\text{-}mset\ (mset\text{-}cls\ D)$ **and**
 $not\text{-}tauto: \neg tautology\ (mset\text{-}cls\ D)$
using $bt\ inv$ **by** $(elim\ backjump\text{-}lE)\ simp$
have $atms\text{-}C': atms\text{-}of\ C' \subseteq atm\text{-}of\ ' (lits\text{-}of\text{-}l\ F)$
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** $tr\text{-}S$
by $(metis\ (no\text{-}types)\ \langle F \models_{as} CNot\ C' \rangle atm\text{-}of\text{-}in\text{-}atm\text{-}of\text{-}set\text{-}iff\text{-}in\text{-}set\text{-}or\text{-}uminus\text{-}in\text{-}set$
 $atms\text{-}of\text{-}def\ in\text{-}CNot\text{-}implies\text{-}uminus(2)\ subsetI)$
qed
then have $atms\text{-}of\ (C' + \{\#L\# \}) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S))$
using $atm\text{-}L\ tr\text{-}S$ **by** $auto$
moreover have $learn: learn\ S\ (add\text{-}cls_{NOT}\ D\ S)$

```

  apply (rule learn.intros)
    apply (rule cls-C)
    using atms-C' atm-L D 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 D S) T
  apply (rule backjump.intros)
  using (F  $\models$  as CNot C') C-cls-S tr-S-CNot-C undef T distinct not-tauto n-d D
  by (auto simp: tr-S state-eqNOT-def simp del: state-simpNOT)
ultimately show ?thesis using D by blast
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 **and** D :: 'cls **where**
 f3: learn S (add-cls_{NOT} D S) \wedge
 backjump (add-cls_{NOT} D S) T \wedge
 atms-of (C' + {#L#}) \subseteq atms-of-mm (clauses_{NOT} S) \cup atm-of ' lits-of-l (trail S) **and**
 D: mset-cls D = C' + {#L#}
 using n-d backjump-l-learn-backjump[OF bt inv] by blast
 then have f4: cdcl_{NOT} S (add-cls_{NOT} D S)
 using n-d c-learn by blast
 have cdcl_{NOT} (add-cls_{NOT} D 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** $st = \text{this}(1)$ **and** $cdcl_{NOT} = \text{this}(2)$ **and** $IH = \text{this}(3)[OF\ \text{this}(4-)]$ **and**
 $inv = \text{this}(4)$ **and** $n-d = \text{this}(5)$
have $cdcl_{NOT}^{**}\ T\ U$
using $cdcl_{NOT}\text{-merged-bj-learn-is-tranclp-cdcl}_{NOT}[OF\ cdcl_{NOT}]\ IH$
 $rtranclp\text{-cdcl}_{NOT}\text{-no-dup}\ inv\ n-d$ **by** *auto*
then have $cdcl_{NOT}^{**}\ S\ U$ **using** IH **by** *fastforce*
moreover have $inv\ U$ **using** $n-d\ IH\ \langle cdcl_{NOT}^{**}\ T\ U \rangle\ rtranclp\text{-cdcl}_{NOT}\text{-inv}$ **by** *blast*
ultimately show $?case$ **using** st **by** *fast*
qed

lemma $rtranclp\text{-cdcl}_{NOT}\text{-merged-bj-learn-is-rtranclp-cdcl}_{NOT}$:
 $cdcl_{NOT}\text{-merged-bj-learn}^{**}\ S\ T \implies inv\ S \implies no\text{-dup}\ (trail\ S) \implies cdcl_{NOT}^{**}\ S\ T$
using $rtranclp\text{-cdcl}_{NOT}\text{-merged-bj-learn-is-rtranclp-cdcl}_{NOT}\text{-and-inv}$ **by** *blast*

lemma $rtranclp\text{-cdcl}_{NOT}\text{-merged-bj-learn-inv}$:
 $cdcl_{NOT}\text{-merged-bj-learn}^{**}\ S\ T \implies inv\ S \implies no\text{-dup}\ (trail\ S) \implies inv\ T$
using $rtranclp\text{-cdcl}_{NOT}\text{-merged-bj-learn-is-rtranclp-cdcl}_{NOT}\text{-and-inv}$ **by** *blast*

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

definition $\mu_{CDCL}'\text{-merged} :: 'v\ \text{literal multiset set} \Rightarrow 'st \Rightarrow nat$ **where**
 $\mu_{CDCL}'\text{-merged}\ A\ T \equiv$
 $((2 + card\ (atms\text{-of}\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-of}\text{-}ms\ A)) - \mu_C'\ A\ T) * 2 + card\ (set\text{-}mset\ (clauses_{NOT}\ T))$

lemma $cdcl_{NOT}\text{-decreasing-measure}'$:
assumes
 $cdcl_{NOT}\text{-merged-bj-learn}\ S\ T$ **and**
 $inv: inv\ S$ **and**
 $atm\text{-}clss: atms\text{-of}\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-of}\text{-}ms\ A$ **and**
 $atm\text{-}trail: atm\text{-of}\ 'lits\text{-of}\text{-}l\ (trail\ S) \subseteq atms\text{-of}\text{-}ms\ A$ **and**
 $n-d: no\text{-dup}\ (trail\ S)$ **and**
 $fin\text{-}A: finite\ A$

shows $\mu_{CDCL}'\text{-merged}\ A\ T < \mu_{CDCL}'\text{-merged}\ A\ S$
using *assms(1)*

proof *induction*

case ($cdcl_{NOT}\text{-merged-bj-learn-decide}_{NOT}\ T$)

have $clauses_{NOT}\ S = clauses_{NOT}\ T$

using $cdcl_{NOT}\text{-merged-bj-learn-decide}_{NOT}.hyps$ **by** *auto*

moreover have

$(2 + card\ (atms\text{-of}\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-of}\text{-}ms\ A))$
 $- \mu_C\ (1 + card\ (atms\text{-of}\text{-}ms\ A))\ (2 + card\ (atms\text{-of}\text{-}ms\ A))\ (trail\text{-}weight\ T)$
 $< (2 + card\ (atms\text{-of}\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-of}\text{-}ms\ A))$
 $- \mu_C\ (1 + card\ (atms\text{-of}\text{-}ms\ A))\ (2 + card\ (atms\text{-of}\text{-}ms\ A))\ (trail\text{-}weight\ S)$

apply (*rule dpll-bj-trail-mes-decreasing-prop*)

using $cdcl_{NOT}\text{-merged-bj-learn-decide}_{NOT}\ fin\text{-}A\ atm\text{-}clss\ atm\text{-}trail\ n-d\ inv$

by (*simp-all add: bj-decide_{NOT} cdcl_{NOT}-merged-bj-learn-decide_{NOT}.hyps*)

ultimately show $?case$

unfolding $\mu_{CDCL}'\text{-merged-def}\ \mu_C'\text{-def}$ **by** *simp*

next

case ($cdcl_{NOT}\text{-merged-bj-learn-propagate}_{NOT}\ T$)

have $clauses_{NOT}\ S = clauses_{NOT}\ T$

using $cdcl_{NOT}\text{-merged-bj-learn-propagate}_{NOT}.hyps$

by (*simp add: bj-propagate_{NOT} inv dpll-bj-clauses*)

moreover have
 $(2 + \text{card}(\text{atms-of-ms } A)) \wedge (1 + \text{card}(\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card}(\text{atms-of-ms } A)) (2 + \text{card}(\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $< (2 + \text{card}(\text{atms-of-ms } A)) \wedge (1 + \text{card}(\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card}(\text{atms-of-ms } A)) (2 + \text{card}(\text{atms-of-ms } A)) (\text{trail-weight } S)$
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 $\text{card}(\text{set-mset}(\text{clauses}_{NOT} T)) < \text{card}(\text{set-mset}(\text{clauses}_{NOT} S))$
using $\langle \text{forget}_{NOT} S T \rangle$ **by** (*metis card-Diff1-less clauses-remove-cls_{NOT} finite-set-mset*
forget_{NOT}.cases in-clss-mset-clss linear set-mset-minus-replicate-mset(1) state-eq_{NOT}-def)
moreover
have *trail S = trail T*
using $\langle \text{forget}_{NOT} S T \rangle$ **by** (*auto elim: forget_{NOT}E*)
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 D* **where**
learn: learn S (add-cls_{NOT} D S) and
bj: backjump (add-cls_{NOT} D S) T and
atms-C: atms-of (C' + {#L#}) \subseteq atms-of-mm (clauses_{NOT} S) \cup atm-of ' (lits-of-l (trail S)) and
D: mset-cls D = C' + {#L#}
using *bj-l inv backjump-l-learn-backjump n-d atm-clss atm-trail* **by** *meson*
have *card-T-S: card (set-mset (clauses_{NOT} T)) \leq 1 + card (set-mset (clauses_{NOT} S))*
using *bj-l inv* **by** (*force elim!: backjump-lE simp: card-insert-if*)
have
 $((2 + \text{card}(\text{atms-of-ms } A)) \wedge (1 + \text{card}(\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card}(\text{atms-of-ms } A)) (2 + \text{card}(\text{atms-of-ms } A)) (\text{trail-weight } T))$
 $< ((2 + \text{card}(\text{atms-of-ms } A)) \wedge (1 + \text{card}(\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card}(\text{atms-of-ms } A)) (2 + \text{card}(\text{atms-of-ms } A))$
 $(\text{trail-weight}(\text{add-cls}_{NOT} D S)))$
apply (rule *dpll-bj-trail-mes-decreasing-prop*)
using *bj bj-backjump* **apply** *blast*
using *cdcl_{NOT}.c-learn cdcl_{NOT}-inv inv learn* **apply** *blast*
using *atms-C atm-clss atm-trail D* **apply** (*simp add: n-d*) **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 $card\text{-}T\text{-}S$ **unfolding** $\mu_{CDCL}'\text{-merged-def}$ $\mu_C'\text{-def}$ **by** $linarith$
qed

lemma $wf\text{-}cdcl_{NOT}\text{-merged-bj-learn}$:

assumes

$fin\text{-}A$: $finite\ A$

shows $wf\ \{(T, S)\}$.

$(inv\ S \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ ' lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
 $\wedge no\text{-}dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}$

apply $(rule\ wfP\text{-}if\text{-}measure[of\ -\ -\ \mu_{CDCL}'\text{-merged}\ A])$

using $cdcl_{NOT}\text{-decreasing-measure}'\ fin\text{-}A$ **by** $simp$

lemma $trancpl\text{-}cdcl_{NOT}\text{-}cdcl_{NOT}\text{-}trancpl$:

assumes

$cdcl_{NOT}\text{-merged-bj-learn}^{++}\ S\ T$ **and**

inv : $inv\ S$ **and**

$atm\text{-}clss$: $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**

$atm\text{-}trail$: $atm\text{-}of\ ' lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**

$n\text{-}d$: $no\text{-}dup\ (trail\ S)$ **and**

$fin\text{-}A[simp]$: $finite\ A$

shows $(T, S) \in \{(T, S)\}$.

$(inv\ S \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ ' lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
 $\wedge no\text{-}dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}^+ \text{ (is - } \in ?P^+)$

using $assms(1)$

proof $(induction\ rule: trancpl\text{-}induct)$

case $base$

then show $?case$ **using** $n\text{-}d\ atm\text{-}clss\ atm\text{-}trail\ inv$ **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 $(rule\ rtrancpl\text{-}cdcl_{NOT}\text{-merged-bj-learn-is-rtrancpl}\text{-}cdcl_{NOT})$

using $st\ cdcl_{NOT}\ inv\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail\ inv$ **by** $auto$

have $inv\ T$

apply $(rule\ rtrancpl\text{-}cdcl_{NOT}\text{-merged-bj-learn-inv)$

using $inv\ st\ cdcl_{NOT}\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail\ inv$ **by** $auto$

moreover have $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ T) \subseteq atms\text{-}of\text{-}ms\ A$

using $rtrancpl\text{-}cdcl_{NOT}\text{-trail-clauses-bound}[OF\ \langle cdcl_{NOT}^{**}\ S\ T \rangle\ inv\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail]$
by $fast$

moreover have $atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ T)) \subseteq atms\text{-}of\text{-}ms\ A$

using $rtrancpl\text{-}cdcl_{NOT}\text{-trail-clauses-bound}[OF\ \langle cdcl_{NOT}^{**}\ S\ T \rangle\ inv\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail]$
by $fast$

moreover have $no\text{-}dup\ (trail\ T)$

using $rtrancpl\text{-}cdcl_{NOT}\text{-no-dup}[OF\ \langle cdcl_{NOT}^{**}\ S\ T \rangle\ inv\ n\text{-}d]$ **by** $fast$

ultimately have $(U, T) \in ?P$

using $cdcl_{NOT}$ **by** $auto$

then show $?case$ **using** IH **by** $(simp\ add: trancpl\text{-}into\text{-}trancpl2)$

qed

lemma $wf\text{-}trancpl\text{-}cdcl_{NOT}\text{-merged-bj-learn}$:

assumes $finite\ A$

shows $wf\ \{(T, S)\}$.

$(inv\ S \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ ' lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$

```

   $\wedge$  no-dup (trail S))
   $\wedge$  cdclNOT-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 - - - -  $\langle$ finite A $\rangle$ ] by auto

```

```

lemma backjump-no-step-backjump-l:
  backjump S T  $\implies$  inv S  $\implies$   $\neg$ no-step backjump-l S
apply (elim backjumpE)
apply (rule bj-merge-can-jump)
  apply auto[7]
by blast

```

```

lemma cdclNOT-merged-bj-learn-final-state:
  fixes A :: 'v literal multiset set and S T :: 'st
  assumes
    n-s: no-step cdclNOT-merged-bj-learn S and
    atms-S: atms-of-mm (clausesNOT S)  $\subseteq$  atms-of-ms A and
    atms-trail: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-ms A and
    n-d: no-dup (trail S) and
    finite A and
    inv: inv S and
    decomp: all-decomposition-implies-m (clausesNOT S) (get-all-marked-decomposition (trail S))
  shows unsatisfiable (set-mset (clausesNOT S))
     $\vee$  (trail S  $\models$ asm clausesNOT S  $\wedge$  satisfiable (set-mset (clausesNOT S)))

```

```

proof -
  let ?N = set-mset (clausesNOT S)
  let ?M = trail S
  consider
    (sat) satisfiable ?N and ?M  $\models$ as ?N
  | (sat') satisfiable ?N and  $\neg$  ?M  $\models$ as ?N
  | (unsat) unsatisfiable ?N
  by auto
  then show ?thesis
  proof cases
    case sat' note sat = this(1) and M = this(2)
    obtain C where C  $\in$  ?N and  $\neg$ ?M  $\models$ a C using M unfolding true-annots-def by auto
    obtain I :: 'v literal set where
      I  $\models$ s ?N and
      cons: consistent-interp I and
      tot: total-over-m I ?N and
      atm-I-N: atm-of 'I  $\subseteq$  atms-of-ms ?N
    using sat unfolding satisfiable-def-min by auto
    let ?I = I  $\cup$  {P | P. P  $\in$  lits-of-l ?M  $\wedge$  atm-of P  $\notin$  atm-of 'I}
    let ?O = { $\{\#lit\text{-of } L\# \}$  | L. is-marked L  $\wedge$  L  $\in$  set ?M  $\wedge$  atm-of (lit-of L)  $\notin$  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  $\cup$  unmark-l ?M)
      using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
      by (fastforce simp: image-iff)
    have {P | P. P  $\in$  lits-of-l ?M  $\wedge$  atm-of P  $\notin$  atm-of 'I}  $\models$ s ?O
      using (I  $\models$ s ?N) atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)

```



```

then have  $I'-N: ?I \models_s ?N \cup ?O$ 
  using  $\langle I \models_s ?N \rangle$  true-clss-union-increase by force
have  $tot': total-over-m ?I (?N \cup ?O)$ 
  using atm-I-N tot unfolding total-over-m-def total-over-set-def
  by (force simp: image-iff lits-of-def dest!: is-marked-ex-Marked)

have atms-N-M: atms-of-ms ?N  $\subseteq$  atm-of ' lits-of-l ?M
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $l :: 'v$  where
     $l-N: l \in atms-of-ms ?N$  and
     $l-M: l \notin atm-of ' lits-of-l ?M$ 
  by auto
have undefined-lit ?M (Pos l)
  using  $l-M$  by (metis Marked-Propagated-in-iff-in-lits-of-l
    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  $?M \models_{as} CNot\ C$ 
  by (metis atms-N-M  $\langle C \in ?N \rangle \langle \neg ?M \models_a C \rangle$  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  $\exists l \in set\ ?M. is-marked\ l$ 
proof (rule ccontr)
  let  $?O = \{\{\#lit-of\ L\# \mid L. is-marked\ L \wedge L \in set\ ?M \wedge atm-of\ (lit-of\ L) \notin atms-of-ms\ ?N\}\}$ 
  have  $\vartheta[iff]: \bigwedge I. total-over-m\ I\ (?N \cup ?O \cup unmark-l\ ?M)$ 
     $\longleftrightarrow total-over-m\ I\ (?N \cup unmark-l\ ?M)$ 
  unfolding total-over-set-def total-over-m-def atms-of-ms-def by auto
  assume  $\neg ?thesis$ 
  then have  $[simp]: \{\{\#lit-of\ L\# \mid L. is-marked\ L \wedge L \in set\ ?M\}\}$ 
     $= \{\{\#lit-of\ L\# \mid L. is-marked\ L \wedge L \in set\ ?M \wedge atm-of\ (lit-of\ L) \notin atms-of-ms\ ?N\}\}$ 
  by auto
  then have  $?N \cup ?O \models_{ps} unmark-l\ ?M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

  then have  $?I \models_s unmark-l\ ?M$ 
    using cons-I' I'-N tot-I'  $\langle ?I \models_s ?N \cup ?O \rangle$  unfolding  $\vartheta$  true-clss-clss-def by blast
  then have  $lits-of-l\ ?M \subseteq ?I$ 
    unfolding true-clss-def lits-of-def by auto
  then have  $?M \models_{as} ?N$ 
    using  $I'-N \langle C \in ?N \rangle \langle \neg ?M \models_a C \rangle$  cons-I' atms-N-M
    by (meson (trail S  $\models_{as} CNot\ C$ ) consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
      true-annots-def true-clss-mono-set-mset-l true-clss-def)
  then show False using  $M$  by fast
qed

from List.split-list-first-propE[OF this] obtain  $K :: 'v$  literal and  $d :: unit$  and
 $F\ F' :: ('v, unit, unit) marked-lit\ list$  where
 $M-K: ?M = F' @ Marked\ K\ () \# F$  and
 $nm: \forall f \in set\ F'. \neg is-marked\ f$ 
  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 unmark-l ?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 unmark-l ?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' ∪ unmark-l ?M =
      ?N ∪ unmark-l ?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 {a ∈ set (trail S). is-marked a ∧ a ≠ Marked K ()} =
      set (trail S) ∩ {L. is-marked L ∧ L ≠ Marked K ()}
    by auto
    then 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  $\neg I \models_s ?C'$ 
    using  $\langle ?N \cup ?C' \models_{ps} \{\{\#\}\} \rangle$  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  $I \models$  image-mset uminus  $?C + \{\# - K\#$ 
    unfolding true-clss-def true-cl-def Bex-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  $F \models_{as} CNot$  (image-mset uminus  $?C$ )
    using nm unfolding true-annots-def CNot-def M-K by (auto simp add: lits-of-def)
  ultimately have False
    using bj-merge-can-jump[of S F' K F C -K
      image-mset uminus (image-mset lit-of  $\{\# L : \#$  mset  $?M$ . is-marked  $L \wedge L \neq$  Marked  $K ()\#\}$ )]
       $\langle C \in ?N \rangle$  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-cdclNOT-merged-bj-learn-final-state:
  fixes A :: 'v literal multiset set and S T :: 'st
  assumes
    full: full cdclNOT-merged-bj-learn S T and
    atms-S: atms-of-mm (clausesNOT S)  $\subseteq$  atms-of-ms A and
    atms-trail: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-ms A and
    n-d: no-dup (trail S) and
    finite A and
    inv: inv S and
    decomp: all-decomposition-implies-m (clausesNOT S) (get-all-marked-decomposition (trail S))
  shows unsatisfiable (set-mset (clausesNOT T))
     $\vee$  (trail T  $\models_{asm}$  clausesNOT T  $\wedge$  satisfiable (set-mset (clausesNOT T)))
proof -
  have st: cdclNOT-merged-bj-learn** S T and n-s: no-step cdclNOT-merged-bj-learn T
    using full unfolding full-def by blast+
  then have st: cdclNOT** S T
    using inv rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT-and-inv n-d by auto
  have atms-of-mm (clausesNOT T)  $\subseteq$  atms-of-ms A and atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A
    using rtranclp-cdclNOT-trail-clauses-bound[OF st inv n-d atms-S atms-trail] by blast+
  moreover have no-dup (trail T)
    using rtranclp-cdclNOT-no-dup inv n-d st by blast
  moreover have inv T
    using rtranclp-cdclNOT-inv inv st by blast
  moreover have all-decomposition-implies-m (clausesNOT T) (get-all-marked-decomposition (trail T))
    using rtranclp-cdclNOT-all-decomposition-implies inv st decomp n-d by blast
  ultimately show ?thesis
    using cdclNOT-merged-bj-learn-final-state[of T A]  $\langle$ finite A $\rangle$  n-s by fast
qed
end

```

15.8.1 Instantiations

locale *cdcl_{NOT}-with-backtrack-and-restarts* =
conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt
mset-cls union-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
inv backjump-conds propagate-conds learn-restrictions forget-restrictions
for
mset-cls :: 'cls \Rightarrow 'v clause **and**
union-cls :: 'cls \Rightarrow 'cls \Rightarrow 'cls **and**
insert-cls :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**
remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**
mset-clss :: 'clss \Rightarrow 'v clauses **and**
union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss **and**
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool **and**
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**
raw-clauses :: 'st \Rightarrow 'clss **and**
prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
add-cl_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
remove-cl_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
inv :: 'st \Rightarrow bool **and**
backjump-conds :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool **and**
propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool **and**
learn-restrictions forget-restrictions :: 'cls \Rightarrow 'st \Rightarrow bool
 +
fixes *f* :: nat \Rightarrow nat
assumes
unbounded: unbounded *f* **and** *f-ge-1*: $\bigwedge n. n \geq 1 \implies f\ n \geq 1$ **and**
inv-restart: $\bigwedge S\ T. inv\ S \implies T \sim \text{reduce-trail-to}_{NOT} ([\]::'a\ list)\ S \implies inv\ T$
begin

lemma *bound-inv-inv*:
assumes
inv S **and**
n-d: no-dup (trail *S*) **and**
atms-clss-S-A: atms-of-mm (clauses_{NOT} *S*) \subseteq atms-of-ms *A* **and**
atms-trail-S-A: atm-of ' lits-of-l (trail *S*) \subseteq atms-of-ms *A* **and**
finite A **and**
cdcl_{NOT}: cdcl_{NOT} *S T*
shows
atms-of-mm (clauses_{NOT} *T*) \subseteq atms-of-ms *A* **and**
atm-of ' lits-of-l (trail *T*) \subseteq atms-of-ms *A* **and**
finite A
proof –
have cdcl_{NOT} *S T*
using <inv *S*> cdcl_{NOT} **by** linarith
then have atms-of-mm (clauses_{NOT} *T*) \subseteq atms-of-mm (clauses_{NOT} *S*) \cup atm-of ' lits-of-l (trail *S*)
using <inv *S*>
by (meson conflict-driven-clause-learning-ops.cdcl_{NOT}-atms-of-ms-clauses-decreasing
 conflict-driven-clause-learning-ops-axioms n-d)
then show atms-of-mm (clauses_{NOT} *T*) \subseteq atms-of-ms *A*
using atms-clss-S-A atms-trail-S-A **by** blast

```

next
  show  $\text{atm-of } \langle \text{lits-of-l } (\text{trail } T) \subseteq \text{atms-of-ms } A$ 
    by ( $\text{meson } \langle \text{inv } S \rangle \text{atms-clss-}S\text{-}A \text{atms-trail-}S\text{-}A \text{cdcl}_{NOT} \text{cdcl}_{NOT}\text{-atms-in-trail-in-set } n\text{-}d$ )
next
  show  $\text{finite } A$ 
    using  $\langle \text{finite } A \rangle$  by  $\text{simp}$ 
qed

sublocale  $\text{cdcl}_{NOT}\text{-increasing-restarts-ops } \lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}]) \ S \ \text{cdcl}_{NOT} \ f$ 
 $\lambda A \ S. \ \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } \langle \text{lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge$ 
 $\text{finite } A$ 
 $\mu_{CDCL}' \lambda S. \ \text{inv } S \wedge \text{no-dup } (\text{trail } S)$ 
 $\mu_{CDCL}'\text{-bound}$ 
apply  $\text{unfold-locales}$ 
  apply ( $\text{simp add: unbounded}$ )
  using  $f\text{-ge-1}$  apply  $\text{force}$ 
  using  $\text{bound-inv-inv}$  apply  $\text{meson}$ 
  apply ( $\text{rule cdcl}_{NOT}\text{-decreasing-measure'; simp}$ )
  apply ( $\text{rule rtranclp-cdcl}_{NOT}\text{-}\mu_{CDCL}'\text{-bound; simp}$ )
  apply ( $\text{rule rtranclp-}\mu_{CDCL}'\text{-bound-decreasing; simp}$ )
  apply  $\text{auto}[]$ 
  apply  $\text{auto}[]$ 
  using  $\text{cdcl}_{NOT}\text{-inv cdcl}_{NOT}\text{-no-dup}$  apply  $\text{blast}$ 
  using  $\text{inv-restart}$  apply  $\text{auto}[]$ 
done

lemma  $\text{cdcl}_{NOT}\text{-with-restart-}\mu_{CDCL}'\text{-le-}\mu_{CDCL}'\text{-bound:}$ 
  assumes
     $\text{cdcl}_{NOT}: \text{cdcl}_{NOT}\text{-restart } (T, a) (V, b) \text{ and}$ 
     $\text{cdcl}_{NOT}\text{-inv:}$ 
     $\text{inv } T$ 
     $\text{no-dup } (\text{trail } T) \text{ and}$ 
     $\text{bound-inv:}$ 
     $\text{atms-of-mm } (\text{clauses}_{NOT} \ T) \subseteq \text{atms-of-ms } A$ 
     $\text{atm-of } \langle \text{lits-of-l } (\text{trail } T) \subseteq \text{atms-of-ms } A$ 
     $\text{finite } A$ 
  shows  $\mu_{CDCL}' A \ V \leq \mu_{CDCL}'\text{-bound } A \ T$ 
  using  $\text{cdcl}_{NOT}\text{-inv bound-inv}$ 
proof ( $\text{induction rule: cdcl}_{NOT}\text{-with-restart-induct}[OF \ \text{cdcl}_{NOT}]$ )
  case ( $1 \ m \ S \ T \ n \ U$ ) note  $U = \text{this}(3)$ 
  show ?case
    apply ( $\text{rule rtranclp-cdcl}_{NOT}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}[\text{of } S \ T]$ )
    using  $\langle (\text{cdcl}_{NOT} \ \widetilde{\sim} \ m) \ S \ T \rangle$  apply ( $\text{fastforce dest!: relpowp-imp-rtranclp}$ )
    using  $1$  by  $\text{auto}$ 
next
  case ( $2 \ S \ T \ n$ ) note  $\text{full} = \text{this}(2)$ 
  show ?case
    apply ( $\text{rule rtranclp-cdcl}_{NOT}\text{-}\mu_{CDCL}'\text{-bound}$ )
    using  $\text{full } 2$  unfolding  $\text{full1-def}$  by  $\text{force+}$ 
qed

lemma  $\text{cdcl}_{NOT}\text{-with-restart-}\mu_{CDCL}'\text{-bound-le-}\mu_{CDCL}'\text{-bound:}$ 
  assumes
     $\text{cdcl}_{NOT}: \text{cdcl}_{NOT}\text{-restart } (T, a) (V, b) \text{ and}$ 
     $\text{cdcl}_{NOT}\text{-inv:}$ 

```

```

    inv T
    no-dup (trail T) and
    bound-inv:
      atms-of-mm (clausesNOT T)  $\subseteq$  atms-of-ms A
      atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A
      finite A
    shows  $\mu_{CDCL}'$ -bound A  $V \leq \mu_{CDCL}'$ -bound A T
    using cdclNOT-inv bound-inv
  proof (induction rule: cdclNOT-with-restart-induct[OF cdclNOT])
    case (1 m S T n U) note U = this(3)
    have  $\mu_{CDCL}'$ -bound A T  $\leq \mu_{CDCL}'$ -bound A S
    apply (rule rtrancp- $\mu_{CDCL}'$ -bound-decreasing)
    using  $\langle \text{cdcl}_{NOT} \rightsquigarrow m \rangle S T$  apply (fastforce dest: relpowp-imp-rtrancp)
    using 1 by auto
    then show ?case using U unfolding  $\mu_{CDCL}'$ -bound-def by auto
  next
    case (2 S T n) note full = this(2)
    show ?case
    apply (rule rtrancp- $\mu_{CDCL}'$ -bound-decreasing)
    using full 2 unfolding full1-def by force+
qed

sublocale cdclNOT-increasing-restarts - - - - -
  f
   $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}] S$ 
 $\lambda A S. \text{atms-of-mm} (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms} A$ 
 $\wedge \text{atm-of ' lits-of-l} (\text{trail} S) \subseteq \text{atms-of-ms} A \wedge \text{finite} A$ 
 $\mu_{CDCL}' \text{ cdcl}_{NOT}$ 
 $\lambda S. \text{inv} S \wedge \text{no-dup} (\text{trail} S)$ 
 $\mu_{CDCL}'$ -bound
  apply unfold-locales
  using cdclNOT-with-restart- $\mu_{CDCL}'$ -le- $\mu_{CDCL}'$ -bound apply simp
  using cdclNOT-with-restart- $\mu_{CDCL}'$ -bound-le- $\mu_{CDCL}'$ -bound apply simp
  done

lemma cdclNOT-restart-all-decomposition-implies:
  assumes cdclNOT-restart S T and
    inv (fst S) and
    no-dup (trail (fst S))
  all-decomposition-implies-m (clausesNOT (fst S)) (get-all-marked-decomposition (trail (fst S)))
  shows
    all-decomposition-implies-m (clausesNOT (fst T)) (get-all-marked-decomposition (trail (fst T)))
  using assms apply (induction)
  using rtrancp-cdclNOT-all-decomposition-implies by (auto dest!: trancp-into-rtrancp
    simp: full1-def)

lemma rtrancp-cdclNOT-restart-all-decomposition-implies:
  assumes cdclNOT-restart** S T and
    inv: inv (fst S) and
    n-d: no-dup (trail (fst S)) and
    decomp:
      all-decomposition-implies-m (clausesNOT (fst S)) (get-all-marked-decomposition (trail (fst S)))
  shows
    all-decomposition-implies-m (clausesNOT (fst T)) (get-all-marked-decomposition (trail (fst T)))
  using assms(1)

```

proof (*induction rule: rtrancpl-induct*)
case *base*
then show *?case* **using** *decomp* **by** *simp*
next
case (*step T u*) **note** *st = this(1)* **and** *r = this(2)* **and** *IH = this(3)*
have *inv (fst T)*
using *rtrancpl-cdcl_{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[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 $I \models_{\text{sextm clauses}_{\text{NOT}}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm clauses}_{\text{NOT}}} (\text{fst } T)$
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 $I \models_{\text{sextm clauses}_{\text{NOT}}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm clauses}_{\text{NOT}}} (\text{fst } T)$
using *st*

proof (*induction*)

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

next

case (*step T U*) **note** *st = this(1)* **and** *r = this(2)* **and** *IH = this(3)*
have *inv (fst T)*
using *rtrancpl-cdcl_{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-mm (clauses_{NOT} S) \subseteq atms-of-ms A **and**

atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) \subseteq *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A[simp]: *finite A* **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses_{NOT} S*) (*get-all-marked-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses_{NOT} S*))
 \vee (*lits-of-l* (*trail T*) \models_{sextm} *clauses_{NOT} S* \wedge *satisfiable* (*set-mset* (*clauses_{NOT} S*)))
proof –
have *st*: *cdcl_{NOT}-restart*** (*S*, *n*) (*T*, *m*) **and**
n-s: *no-step cdcl_{NOT}-restart* (*T*, *m*)
using *full unfolding full-def* **by** *fast+*
have *binv-T*: *atms-of-mm* (*clauses_{NOT} T*) \subseteq *atms-of-ms A*
atm-of ‘ *lits-of-l* (*trail T*) \subseteq *atms-of-ms A*
using *rtrancpl-cdcl_{NOT}-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 *rtrancpl-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*[*OF st*] *inv n-d* **by** *auto*
moreover have *all-decomposition-implies-m* (*clauses_{NOT} T*) (*get-all-marked-decomposition* (*trail T*))
using *rtrancpl-cdcl_{NOT}-restart-all-decomposition-implies*[*OF st*] *inv n-d*
decomp **by** *auto*
ultimately have *T*: *unsatisfiable* (*set-mset* (*clauses_{NOT} T*))
 \vee (*trail T* \models_{asm} *clauses_{NOT} T* \wedge *satisfiable* (*set-mset* (*clauses_{NOT} T*)))
using *no-step-cdcl_{NOT}-restart-no-step-cdcl_{NOT}*[*of* (*T*, *m*) *A*] *n-s*
cdcl_{NOT}-final-state[*of T A*] **unfolding** *cdcl_{NOT}-NOT-all-inv-def* **by** *auto*
have *eq-sat-S-T*: $\bigwedge I. I \models_{\text{sextm}}$ *clauses_{NOT} S* \longleftrightarrow $I \models_{\text{sextm}}$ *clauses_{NOT} T*
using *rtrancpl-cdcl_{NOT}-restart-sat-ext-iff*[*OF st*] *inv n-d atms-S*
atms-trail **by** *auto*
have *cons-T*: *consistent-interp* (*lits-of-l* (*trail T*))
using *inv-T(1) distinct-consistent-interp* **by** *blast*
consider
(*unsat*) *unsatisfiable* (*set-mset* (*clauses_{NOT} T*))
| (*sat*) *trail T* \models_{asm} *clauses_{NOT} T* **and** *satisfiable* (*set-mset* (*clauses_{NOT} T*))
using *T* **by** *blast*
then show *?thesis*
proof *cases*
case *unsat*
then have *unsatisfiable* (*set-mset* (*clauses_{NOT} 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-l* (*trail T*) \models_{sextm} *clauses_{NOT} S*
using *rtrancpl-cdcl_{NOT}-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_{NOT} S*))
using *cons-T consistent-true-clss-ext-satisfiable* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed
qed
end — end of *cdcl_{NOT}-with-backtrack-and-restarts* locale

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* =


```

cdclNOT-merge-bj-learn mset-cls union-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
 $\lambda C \ C' \ L' \ S \ T. \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \text{backjump-l-cond } C \ C' \ L' \ S \ T$ 
propagate-conds forget-conds inv
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  union-cls :: 'cls  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss :: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  forget-conds :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
+
fixes f :: nat  $\Rightarrow$  nat
assumes
  unbounded: unbounded f and f-ge-1:  $\bigwedge n. n \geq 1 \implies f \ n \geq 1$  and
  inv-restart:  $\bigwedge S \ T. \text{inv } S \implies T \sim \text{reduce-trail-to}_{\text{NOT}} \ \Box \ S \implies \text{inv } T$ 
begin

```

definition *not-simplified-cls* $A = \{\#C \in \# \ A. \text{tautology } C \vee \neg \text{distinct-mset } C\#\}$

lemma *simple-clss-or-not-simplified-cls*:

assumes *atms-of-mm* (*clauses*_{NOT} *S*) \subseteq *atms-of-ms* *A* and

$x \in \# \text{clauses}_{\text{NOT}} \ S$ and *finite* *A*

shows $x \in \text{simple-clss } (\text{atms-of-ms } A) \vee x \in \# \text{not-simplified-cls } (\text{clauses}_{\text{NOT}} \ S)$

proof –

consider

(*simpl*) $\neg \text{tautology } x$ and *distinct-mset* *x*

| (*n-simp*) $\text{tautology } x \vee \neg \text{distinct-mset } x$

by *auto*

then show ?thesis

proof *cases*

case *simpl*

then have $x \in \text{simple-clss } (\text{atms-of-ms } A)$

by (*meson* *assms* *atms-of-atms-of-ms-mono* *atms-of-ms-finite* *simple-clss-mono* *distinct-mset-not-tautology-implies-in-simple-clss* *finite-subset* *subsetCE*)

then show ?thesis **by** *blast*

next

case *n-simp*

then have $x \in \# \text{not-simplified-cls } (\text{clauses}_{\text{NOT}} \ S)$

using $\langle x \in \# \text{clauses}_{\text{NOT}} \ S \rangle$ **unfolding** *not-simplified-cls-def* **by** *auto*

```

    then show ?thesis by blast
qed
qed

lemma cdclNOT-merged-bj-learn-clauses-bound:
  assumes
    cdclNOT-merged-bj-learn  $S$   $T$  and
    inv: inv  $S$  and
    atms-clss: atms-of-mm (clausesNOT  $S$ )  $\subseteq$  atms-of-ms  $A$  and
    atms-trail: atm-of (lits-of-l (trail  $S$ ))  $\subseteq$  atms-of-ms  $A$  and
    n-d: no-dup (trail  $S$ ) and
    fin-A[simp]: finite  $A$ 
  shows set-mset (clausesNOT  $T$ )  $\subseteq$  set-mset (not-simplified-cls (clausesNOT  $S$ ))
     $\cup$  simple-clss (atms-of-ms  $A$ )
  using assms
proof (induction rule: cdclNOT-merged-bj-learn.induct)
  case cdclNOT-merged-bj-learn-decideNOT
  then show ?case using dpll-bj-clauses by (force dest!: simple-clss-or-not-simplified-cls)
next
  case cdclNOT-merged-bj-learn-propagateNOT
  then show ?case using dpll-bj-clauses by (force dest!: simple-clss-or-not-simplified-cls)
next
  case cdclNOT-merged-bj-learn-forgetNOT
  then show ?case using clauses-remove-clNOT unfolding state-eqNOT-def
    by (force elim!: forgetNOT $E$  dest: simple-clss-or-not-simplified-cls)
next
  case (cdclNOT-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 cdclNOT**  $S$   $T$ 
  apply (rule rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT)
  using (backjump-l  $S$   $T$ ) inv cdclNOT-merged-bj-learn.simps n-d by blast+
  have atm-of (lits-of-l (trail  $T$ ))  $\subseteq$  atms-of-ms  $A$ 
  using rtranclp-cdclNOT-trail-clauses-bound[OF (cdclNOT**  $S$   $T$ )] inv atms-trail atms-clss
    n-d by auto
  have atms-of-mm (clausesNOT  $T$ )  $\subseteq$  atms-of-ms  $A$ 
  using rtranclp-cdclNOT-trail-clauses-bound[OF (cdclNOT**  $S$   $T$ )] inv n-d atms-clss atms-trail
    by fast
  moreover have no-dup (trail  $T$ )
  using rtranclp-cdclNOT-no-dup[OF (cdclNOT**  $S$   $T$ )] inv n-d by fast

  obtain  $F' K F L l C' C D$  where
    tr- $S$ : trail  $S$  =  $F' @ \text{Marked } K () \# F$  and
     $T$ :  $T \sim \text{prepend-trail } (\text{Propagated } L l) (\text{reduce-trail-to}_{\text{NOT}} F (\text{add-cl}_{\text{NOT}} D S))$  and
     $C \in \# \text{ clauses}_{\text{NOT}} S$  and
    trail  $S \models_{\text{as}} C \text{Not } C$  and
    undef: undefined-lit  $F L$  and
    clausesNOT  $S \models_{\text{pm}} C' + \{\#L\# \}$  and
     $F \models_{\text{as}} C \text{Not } C'$  and
     $D$ : mset-cls  $D = C' + \{\#L\# \}$  and
    dist: distinct-mset ( $C' + \{\#L\# \}$ ) and
    tauto:  $\neg \text{tautology } (C' + \{\#L\# \})$  and
    backjump-l-cond  $C C' L S T$ 
  using (backjump-l  $S T$ ) apply (elim backjump-lE) by auto

```

have $\text{atms-of } C' \subseteq \text{atm-of } \langle \text{lits-of-l } F \rangle$
using $\langle F \models_{\text{as}} C \text{Not } C' \rangle$ **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 $T \langle \text{atm-of } \langle \text{lits-of-l } (\text{trail } T) \subseteq \text{atms-of-ms } A \rangle \text{ tr-S undef n-d} \rangle$ **by** *auto*
then have $\text{simple-clss } (\text{atms-of } (C' + \{\#L\# \})) \subseteq \text{simple-clss } (\text{atms-of-ms } A)$
apply – **by** (*rule simple-clss-mono*) (*simp-all*)
then have $C' + \{\#L\# \} \in \text{simple-clss } (\text{atms-of-ms } A)$
using *distinct-mset-not-tautology-implies-in-simple-clss[OF dist tauto]*
by *auto*
then show ?*case*
using $T \text{ inv atms-clss undef tr-S n-d D}$ **by** (*force dest!: simple-clss-or-not-simplified-clss*)
qed

lemma *cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl_{NOT}-merged-bj-learn S T*
shows $(\text{not-simplified-clss } (\text{clauses}_{\text{NOT}} T)) \subseteq \# (\text{not-simplified-clss } (\text{clauses}_{\text{NOT}} S))$
using *assms apply induction*
prefer 4
unfolding *not-simplified-clss-def* **apply** (*auto elim!: backjump-lE forget_{NOT}E*)[3]
by (*elim backjump-lE*) *auto*

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl_{NOT}-merged-bj-learn** S T*
shows $(\text{not-simplified-clss } (\text{clauses}_{\text{NOT}} T)) \subseteq \# (\text{not-simplified-clss } (\text{clauses}_{\text{NOT}} S))$
using *assms apply induction*
apply *simp*
by (*drule cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*) *auto*

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound*:
assumes
*cdcl_{NOT}-merged-bj-learn** S T and*
inv S and
atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A and
atm-of $\langle \text{lits-of-l } (\text{trail } S) \rangle \subseteq \text{atms-of-ms } A$ and
n-d: no-dup (trail S) and
finite[simp]: finite A
shows $\text{set-mset } (\text{clauses}_{\text{NOT}} T) \subseteq \text{set-mset } (\text{not-simplified-clss } (\text{clauses}_{\text{NOT}} S))$
 $\cup \text{simple-clss } (\text{atms-of-ms } A)$
using *assms(1–5)*

proof *induction*

case *base*

then show ?*case* **by** (*auto dest!: simple-clss-or-not-simplified-clss*)

next

case (*step T U*) **note** $st = \text{this}(1)$ **and** $\text{cdcl}_{\text{NOT}} = \text{this}(2)$ **and** $IH = \text{this}(3)[\text{OF } \text{this}(4–7)]$ **and**
 $\text{inv} = \text{this}(4)$ **and** $\text{atms-clss-S} = \text{this}(5)$ **and** $\text{atms-trail-S} = \text{this}(6)$ **and** $\text{finite-clss-S} = \text{this}(7)$

have $st': \text{cdcl}_{\text{NOT}}^{**} S T$

using $\text{inv rtranclp-cdcl}_{\text{NOT}}\text{-merged-bj-learn-is-rtranclp-cdcl}_{\text{NOT}}\text{-and-inv } st \text{ n-d}$ **by** *blast*

have $\text{inv } T$

using $\text{inv rtranclp-cdcl}_{\text{NOT}}\text{-merged-bj-learn-inv } st \text{ n-d}$ **by** *blast*

moreover

have $\text{atms-of-mm } (\text{clauses}_{\text{NOT}} T) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } \langle \text{lits-of-l } (\text{trail } T) \rangle \subseteq \text{atms-of-ms } A$

using $\text{rtranclp-cdcl}_{\text{NOT}}\text{-trail-clauses-bound}[\text{OF } st']$ $\text{inv atms-clss-S atms-trail-S n-d}$

by *blast+*

moreover moreover have *no-dup* (*trail T*)
using *rtranclp-cdcl_{NOT}-no-dup*[*OF* $\langle \text{cdcl}_{NOT}^{**} S T \rangle$ *inv n-d*] **by** *fast*
ultimately have *set-mset* (*clauses_{NOT} U*)
 \subseteq *set-mset* (*not-simplified-cls* (*clauses_{NOT} T*)) \cup *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_{NOT} T*))
 \subseteq *set-mset* (*not-simplified-cls* (*clauses_{NOT} 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!*: *simple-clss-or-not-simplified-cls*)
qed

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

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound-card*:

assumes
*cdcl_{NOT}-merged-bj-learn^{**} S T* **and**
inv S **and**
atms-of-mm (*clauses_{NOT} S*) \subseteq *atms-of-ms A* **and**
atm-of $\langle \text{lits-of-l } (\text{trail } S) \rangle \subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows μ_{CDCL}' -*merged A T* \leq μ_{CDCL}' -*bound A S*
proof –
have *set-mset* (*clauses_{NOT} T*) \subseteq *set-mset* (*not-simplified-cls*(*clauses_{NOT} S*))
 \cup *simple-clss* (*atms-of-ms A*)
using *rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound*[*OF assms*] .
moreover have *card* (*set-mset* (*not-simplified-cls*(*clauses_{NOT} S*))
 \cup *simple-clss* (*atms-of-ms A*))
 \leq *card* (*set-mset* (*not-simplified-cls*(*clauses_{NOT} S*))) $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$
by (*meson Nat.le-trans atms-of-ms-finite simple-clss-card card-Un-le finite*
nat-add-left-cancel-le)
ultimately have *card* (*set-mset* (*clauses_{NOT} T*))
 \leq *card* (*set-mset* (*not-simplified-cls*(*clauses_{NOT} S*))) $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$
by (*meson Nat.le-trans atms-of-ms-finite simple-clss-finite card-mono*
finite-UnI finite-set-mset local.finite)
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
 $\lambda A S. \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of } \langle \text{lits-of-l } (\text{trail } S) \rangle \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 μ_{CDCL}' -*merged*
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 μ_{CDCL}' -*bound*
apply *unfold-locales*
using *unbounded apply simp*

```

    using f-ge-1 apply force
    apply (blast dest!: cdclNOT-merged-bj-learn-is-tranclp-cdclNOT tranclp-into-rtranclp
      rtranclp-cdclNOT-trail-clauses-bound)
    apply (simp add: cdclNOT-decreasing-measure^)
    using rtranclp-cdclNOT-merged-bj-learn-clauses-bound-card apply blast
    apply (drule rtranclp-cdclNOT-merged-bj-learn-not-simplified-decreasing)
    apply (auto dest!: simp: card-mono set-mset-mono )[]
  apply simp
  apply auto[]
  using cdclNOT-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**
 $atms-of-mm$ ($clauses_{NOT}$ (fst T)) \subseteq $atms-of-ms$ A **and**
 $atm-of$ ' $lits-of-l$ ($trail$ (fst T)) \subseteq $atms-of-ms$ A **and**
 $finite$ A

shows μ_{CDCL}' -merged A (fst V) \leq μ_{CDCL}' -bound A (fst T)

using $assms$

proof $induction$

case ($restart-full$ S T n)

show ?case

unfolding $fst-conv$

apply ($rule$ $rtranclp-cdcl_{NOT}$ -merged-bj-learn-clauses-bound-card)

using $restart-full$ **unfolding** $full1-def$ **by** ($force$ $dest!$: $tranclp-into-rtranclp$)+

next

case ($restart-step$ m S T n U) **note** $st = this(1)$ **and** $U = this(3)$ **and** $inv = this(4)$ **and**
 $n-d = this(5)$ **and** $atms-clss = this(6)$ **and** $atms-trail = this(7)$ **and** $finite = this(8)$

then have st' : $cdcl_{NOT}$ -merged-bj-learn** S T

by ($blast$ $dest$: $relpowp-imp-rtranclp$)

then have st'' : $cdcl_{NOT}$ ** S T

using inv $n-d$ **apply** – **by** ($rule$ $rtranclp-cdcl_{NOT}$ -merged-bj-learn-is-rtranclp-cdcl_{NOT}) $auto$

have inv T

apply ($rule$ $rtranclp-cdcl_{NOT}$ -merged-bj-learn-inv)

using inv st' $n-d$ **by** $auto$

then have inv U

using U **by** ($auto$ $simp$: $inv-restart$)

have $atms-of-mm$ ($clauses_{NOT}$ T) \subseteq $atms-of-ms$ A

using $rtranclp-cdcl_{NOT}$ -trail-clauses-bound[OF st''] inv $atms-clss$ $atms-trail$ $n-d$
by $simp$

then have $atms-of-mm$ ($clauses_{NOT}$ U) \subseteq $atms-of-ms$ A

using U **by** $simp$

have $not-simplified-cls$ ($clauses_{NOT}$ U) $\subseteq\#$ $not-simplified-cls$ ($clauses_{NOT}$ T)

using $\langle U \sim reduce-trail-to_{NOT} \rangle T$ **by** $auto$

moreover have $not-simplified-cls$ ($clauses_{NOT}$ T) $\subseteq\#$ $not-simplified-cls$ ($clauses_{NOT}$ S)

apply ($rule$ $rtranclp-cdcl_{NOT}$ -merged-bj-learn-not-simplified-decreasing)

using $\langle cdcl_{NOT}$ -merged-bj-learn $\sim m$ S $T \rangle$ **by** ($auto$ $dest!$: $relpowp-imp-rtranclp$)

ultimately have $U-S$: $not-simplified-cls$ ($clauses_{NOT}$ U) $\subseteq\#$ $not-simplified-cls$ ($clauses_{NOT}$ S)
by $auto$

have ($set-mset$ ($clauses_{NOT}$ U))

$\subseteq \text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} U)) \cup \text{simple-clss } (\text{atms-of-ms } A)$
apply (rule rtrancp-cdcl_{NOT}-merged-bj-learn-clauses-bound)
 apply simp
 using $\langle \text{inv } U \rangle$ **apply** simp
 using $\langle \text{atms-of-mm } (\text{clauses}_{NOT} U) \subseteq \text{atms-of-ms } A \rangle$ **apply** simp
 using U **apply** simp
 using U **apply** simp
 using finite **apply** simp
 done
then have f1: $\text{card } (\text{set-mset } (\text{clauses}_{NOT} U)) \leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} U)) \cup \text{simple-clss } (\text{atms-of-ms } A))$
 by (simp add: simple-clss-finite card-mono local.finite)

moreover have $\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} U)) \cup \text{simple-clss } (\text{atms-of-ms } A)$
 $\subseteq \text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S)) \cup \text{simple-clss } (\text{atms-of-ms } A)$
 using $U-S$ **by** auto
then have f2:
 $\text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} U)) \cup \text{simple-clss } (\text{atms-of-ms } A))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S)) \cup \text{simple-clss } (\text{atms-of-ms } A))$
 by (simp add: simple-clss-finite card-mono local.finite)

moreover have $\text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S)) \cup \text{simple-clss } (\text{atms-of-ms } A))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S))) + \text{card } (\text{simple-clss } (\text{atms-of-ms } A))$
 using card-Un-le **by** blast
moreover have $\text{card } (\text{simple-clss } (\text{atms-of-ms } A)) \leq 3 \wedge \text{card } (\text{atms-of-ms } A)$
 using atms-of-ms-finite simple-clss-card local.finite **by** blast
ultimately have $\text{card } (\text{set-mset } (\text{clauses}_{NOT} U))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S))) + 3 \wedge \text{card } (\text{atms-of-ms } A)$
 by linarith
then show ?case **unfolding** μ_{CDCL}' -merged-def **by** auto
qed

lemma cdcl_{NOT}-restart- μ_{CDCL}' -bound-le- μ_{CDCL}' -bound:
assumes
 cdcl_{NOT}-restart $T V$ **and**
 no-dup (trail (fst T)) **and**
 inv (fst T) **and**
 fin: finite A
shows μ_{CDCL}' -bound A (fst V) $\leq \mu_{CDCL}'$ -bound A (fst T)
using assms(1-3)

proof induction
case (restart-full $S T n$)
have $\text{not-simplified-cls } (\text{clauses}_{NOT} T) \subseteq \# \text{not-simplified-cls } (\text{clauses}_{NOT} S)$
 apply (rule rtrancp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing)
 using $\langle \text{full1 cdcl}_{NOT}\text{-merged-bj-learn } S T \rangle$ **unfolding** full1-def
 by (auto dest: trancp-into-rtrancp)
then show ?case **by** (auto simp: card-mono set-mset-mono)

next
case (restart-step $m S T n U$) **note** $st = \text{this}(1)$ **and** $U = \text{this}(3)$ **and** $n-d = \text{this}(4)$ **and** $inv = \text{this}(5)$
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 inv $n-d$ **apply** - **by** (rule rtrancp-cdcl_{NOT}-merged-bj-learn-is-rtrancp-cdcl_{NOT}) auto

```

have inv T
  apply (rule rtrancpl-cdclNOT-merged-bj-learn-inv)
  using inv st' n-d by auto
then have inv U
  using U by (auto simp: inv-restart)
have not-simplified-cl (clausesNOT U)  $\subseteq$ # not-simplified-cl (clausesNOT T)
  using  $\langle U \sim \text{reduce-trail-to}_{\text{NOT}} [] T \rangle$  by auto
moreover have not-simplified-cl (clausesNOT T)  $\subseteq$ # not-simplified-cl (clausesNOT S)
  apply (rule rtrancpl-cdclNOT-merged-bj-learn-not-simplified-decreasing)
  using  $\langle (\text{cdcl}_{\text{NOT}}\text{-merged-bj-learn } \widetilde{m}) S T \rangle$  by (auto dest!: relpowp-imp-rtrancpl)
ultimately have U-S: not-simplified-cl (clausesNOT U)  $\subseteq$ # not-simplified-cl (clausesNOT S)
  by auto
then show ?case by (auto simp: card-mono set-mset-mono)
qed

```

```

sublocale cdclNOT-increasing-restarts - - - - - f
   $\lambda S T. T \sim \text{reduce-trail-to}_{\text{NOT}} ([::'a \text{ list}] S)$ 
   $\lambda A S. \text{atms-of-mm } (\text{clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A$ 
   $\wedge \text{atm-of } ' \text{ lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$ 
   $\mu_{\text{CDCL}}'\text{-merged } \text{cdcl}_{\text{NOT}}\text{-merged-bj-learn}$ 
   $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$ 
   $\lambda A T. ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$ 
   $+ \text{card } (\text{set-mset } (\text{not-simplified-cl}(\text{clauses}_{\text{NOT}} T)))$ 
   $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$ 
  apply unfold-locales
  using cdclNOT-restart- $\mu_{\text{CDCL}}'\text{-merged-le-}\mu_{\text{CDCL}}'\text{-bound}$  apply force
  using cdclNOT-restart- $\mu_{\text{CDCL}}'\text{-bound-le-}\mu_{\text{CDCL}}'\text{-bound}$  by fastforce

```

```

lemma cdclNOT-restart-eq-sat-iff:
  assumes
    cdclNOT-restart S T and
    no-dup (trail (fst S))
    inv (fst S)
  shows  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T)$ 
  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 rtrancpl-cdclNOT-bj-sat-ext-iff restart-full.prem1,2
    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  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T$ 
    using rtrancpl-cdclNOT-bj-sat-ext-iff restart-step.prem1,2
    rtrancpl-cdclNOT-merged-bj-learn-is-rtrancpl-cdclNOT by auto
  moreover have  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} U$ 
    using restart-step.hyps(3) by auto
  ultimately show ?case by auto
qed

```

lemma *rtrancpl-cdcl_{NOT}-restart-eq-sat-iff*:

assumes

*cdcl_{NOT}-restart*** *S T* **and**

inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*))

shows $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T)$

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-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv* **using** *st inv n-d* **by** *blast*+

then have $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } U)$

using *cdcl_{NOT}-restart-eq-sat-iff cdcl* **by** *blast*

then show ?*case* **using** *IH* **by** *blast*

qed

lemma *cdcl_{NOT}-restart-all-decomposition-implies-m*:

assumes

cdcl_{NOT}-restart *S T* **and**

inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**

all-decomposition-implies-m (*clauses_{NOT}* (*fst S*))

(*get-all-marked-decomposition* (*trail* (*fst S*)))

shows *all-decomposition-implies-m* (*clauses_{NOT}* (*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*: *cdcl_{NOT}-merged-bj-learn*** *S T* **and**

n-s: *no-step cdcl_{NOT}-merged-bj-learn T*

using *full unfolding full1-def* **by** (*fast dest*: *trancpl-into-rtrancpl*)+

have *st'*: *cdcl_{NOT}*** *S T*

using *inv rtrancpl-cdcl_{NOT}-merged-bj-learn-is-rtrancpl-cdcl_{NOT}-and-inv st n-d* **by** *auto*

have *inv T*

using *rtrancpl-cdcl_{NOT}-cdcl_{NOT}-inv[OF st]* *inv n-d* **by** *auto*

then show ?*case*

using *rtrancpl-cdcl_{NOT}-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

*cdcl_{NOT}-restart*** *S T* **and**

inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**

decomp: *all-decomposition-implies-m* (*clauses_{NOT}* (*fst S*))

(*get-all-marked-decomposition* (*trail* (*fst S*)))

shows *all-decomposition-implies-m* (*clauses_{NOT}* (*fst T*))

(*get-all-marked-decomposition* (*trail* (*fst T*)))

using *assms*

proof (*induction*)


```

case base
then show ?case using decomp by simp
next
case (step T U) note st = this(1) and cdcl = this(2) and IH = this(3)[OF this(4-)] and
  inv = this(4) and n-d = this(5) and decomp = this(6)
have inv (fst T) and no-dup (trail (fst T))
  using rtranclp-cdclNOT-with-restart-cdclNOT-inv using st inv n-d by blast+
then show ?case
  using cdclNOT-restart-all-decomposition-implies-m[OF cdcl] IH by auto
qed

lemma full-cdclNOT-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 (clausesNOT (fst S))
    (get-all-marked-decomposition (trail (fst S))) and
  atms-cl: atms-of-mm (clausesNOT (fst S))  $\subseteq$  atms-of-ms A and
  atms-trail: atm-of ' lits-of-l (trail (fst S))  $\subseteq$  atms-of-ms A and
  fin: finite A
shows unsatisfiable (set-mset (clausesNOT (fst S)))
   $\vee$  lits-of-l (trail (fst T))  $\models_{\text{sextm}}$  clausesNOT (fst S)  $\wedge$  satisfiable (set-mset (clausesNOT (fst S)))
proof -
have inv-T: inv (fst T) and n-d-T: no-dup (trail (fst T))
  using rtranclp-cdclNOT-with-restart-cdclNOT-inv using full inv n-d unfolding full-def by blast+
moreover have
  atms-cl-T: atms-of-mm (clausesNOT (fst T))  $\subseteq$  atms-of-ms A and
  atms-trail-T: atm-of ' lits-of-l (trail (fst T))  $\subseteq$  atms-of-ms A
  using rtranclp-cdclNOT-with-restart-bound-inv[of S T A] full atms-cl 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 (clausesNOT (fst T))
  (get-all-marked-decomposition (trail (fst T)))
  using rtranclp-cdclNOT-restart-all-decomposition-implies-m[of S T] inv n-d decomp
  full unfolding full-def by auto
ultimately have unsatisfiable (set-mset (clausesNOT (fst T)))
   $\vee$  trail (fst T)  $\models_{\text{asm}}$  clausesNOT (fst T)  $\wedge$  satisfiable (set-mset (clausesNOT (fst T)))
  apply -
  apply (rule cdclNOT-merged-bj-learn-final-state)
  using atms-cl-T atms-trail-T fin n-d-T fin inv-T by blast+
then consider
  (unsat) unsatisfiable (set-mset (clausesNOT (fst T)))
  | (sat) trail (fst T)  $\models_{\text{asm}}$  clausesNOT (fst T) and satisfiable (set-mset (clausesNOT (fst T)))
  by auto
then show unsatisfiable (set-mset (clausesNOT (fst S)))
   $\vee$  lits-of-l (trail (fst T))  $\models_{\text{sextm}}$  clausesNOT (fst S)  $\wedge$  satisfiable (set-mset (clausesNOT (fst S)))
proof cases
  case unsat
  then have unsatisfiable (set-mset (clausesNOT (fst S)))

```

```

    unfolding satisfiable-def apply auto
    using rtrancplp-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-l (trail (fst T))  $\models_{\text{sextm}}$  clausesNOT (fst T)
  using true-clss-imp-true-clss-ext by (auto simp: true-annots-true-clss)
then have lits-of-l (trail (fst T))  $\models_{\text{sextm}}$  clausesNOT (fst S)
  using rtrancplp-cdclNOT-restart-eq-sat-iff[of S T] full inv n-d unfolding full-def by blast
moreover then have satisfiable (set-mset (clausesNOT (fst S)))
  using consistent-true-clss-ext-satisfiable distinct-consistent-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 = [] clausesNOT S = N and
  full: full cdclNOT-restart (S, 0) T and
  inv: inv S
shows unsatisfiable (set-mset N)
   $\vee$  lits-of-l (trail (fst T))  $\models_{\text{sextm}}$  N  $\wedge$  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

```

16 DPLL as an instance of NOT

16.1 DPLL with simple backtrack

We are using a concrete couple instead of an abstract state.

```

locale dppll-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  $\models_{\text{as}}$  CNot D  $\Longrightarrow$  backtrack S (Propagated (- (lit-of L)) () # M, snd S)

inductive-cases backtrackE[elim]: backtrack (M, N) (M', N')
lemma backtrack-is-backjump:
fixes M M' :: ('v, unit, 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' @ Marked K () # F  $\wedge$ 

```

$M' = \text{Propagated } L \# F \wedge N = N' \wedge C \in \# N \wedge F' @ \text{Marked } K \# F \models_{as} CNot \ C \wedge$
 $\text{undefined-lit } F \ L \wedge \text{atm-of } L \in \text{atms-of-mm } N \cup \text{atm-of } ' \text{ lits-of-l } (F' @ \text{Marked } K \# F) \wedge$
 $N \models_{pm} C' + \{\#L\} \wedge F \models_{as} CNot \ C'$

proof –

let $?S = (M, N)$

let $?T = (M', N')$

obtain $F \ F' \ P \ L \ D$ **where**

$b\text{-sp}$: $\text{backtrack-split } M = (F', L \# F)$ **and**

$is\text{-marked } L$ **and**

$D \in \# \text{snd } ?S$ **and**

$M \models_{as} CNot \ D$ **and**

bt : $\text{backtrack } ?S \ (\text{Propagated } (- \ (\text{lit-of } L)) \ P \ \# \ F, N)$ **and**

M' : $M' = \text{Propagated } (- \ (\text{lit-of } L)) \ P \ \# \ F$ **and**

$[simp]$: $N' = N$

using $\text{backtrackE}[OF \ \text{backtrack}]$ **by** $(metis \ \text{backtrack} \ \text{fstI} \ \text{sndI})$

let $?K = \text{lit-of } L$

let $?C = \text{image-mset lit-of } \{\#K \in \#mset \ M. \ is\text{-marked } K \wedge K \neq L\} :: 'v \text{ literal multiset}$

let $?C' = \text{set-mset } (\text{image-mset single } (?C + \{\#?K\}))$

obtain K **where** $L = \text{Marked } K \ ()$ **using** $\langle is\text{-marked } L \rangle$ **by** $(cases \ L) \ \text{auto}$

have M : $M = F' @ \text{Marked } K \ () \ \# \ F$

using $b\text{-sp}$ **by** $(metis \ L \ \text{backtrack-split-list-eq} \ \text{fst-conv} \ \text{snd-conv})$

moreover have $F' @ \text{Marked } K \ () \ \# \ F \models_{as} CNot \ D$

using $\langle M \models_{as} CNot \ D \rangle$ **unfolding** M .

moreover have $\text{undefined-lit } F \ (-?K)$

using $no\text{-dup}$ **unfolding** $M \ L$ **by** $(simp \ \text{add: defined-lit-map})$

moreover have $\text{atm-of } (-K) \in \text{atms-of-mm } N \cup \text{atm-of } ' \text{ lits-of-l } (F' @ \text{Marked } K \# F)$

by $auto$

moreover

have $\text{set-mset } N \cup ?C' \models_{ps} \{\{\#\}\}$

proof –

have A : $\text{set-mset } N \cup ?C' \cup \text{unmark-l } M =$

$\text{set-mset } N \cup \text{unmark-l } M$

unfolding $M \ L$ **by** $auto$

have $\text{set-mset } N \cup \{\{\#\text{lit-of } L\}\} \mid L. \ is\text{-marked } L \wedge L \in \text{set } M\}$

$\models_{ps} \text{unmark-l } M$

using $\text{all-decomposition-implies-propagated-lits-are-implied}[OF \ \text{decomp}]$.

moreover have C' : $?C' = \{\{\#\text{lit-of } L\}\} \mid L. \ is\text{-marked } L \wedge L \in \text{set } M\}$

unfolding $M \ L$ **apply** $standard$

apply $force$

using IntI **by** $auto$

ultimately have $N\text{-C-M}$: $\text{set-mset } N \cup ?C' \models_{ps} \text{unmark-l } M$

by $auto$

have $\text{set-mset } N \cup (\lambda L. \ \{\#\text{lit-of } L\}) \ ' \ (\text{set } M) \models_{ps} \{\{\#\}\}$

unfolding $\text{true-clss-clss-def}$

proof $(intro \ allI \ impI, \ \text{goal-cases})$

case $(1 \ I)$ **note** $tot = \text{this}(1)$ **and** $cons = \text{this}(2)$ **and** $I\text{-N-M} = \text{this}(3)$

have $I \models D$

using $I\text{-N-M} \ \langle D \in \# \text{snd } ?S \rangle$ **unfolding** true-clss-def **by** $auto$

moreover have $I \models_s CNot \ D$

using $\langle M \models_{as} CNot \ D \rangle$ **unfolding** M **by** $(metis \ 1(3) \ \langle M \models_{as} CNot \ D \rangle$

$\text{true-annots-true-clss} \ \text{true-clss-mono-set-mset-l} \ \text{true-clss-def}$

$\text{true-clss-singleton-lit-of-implies-incl} \ \text{true-clss-union})$

ultimately show $?case$ **using** $cons \ \text{consistent-CNot-not}$ **by** $blast$

qed

```

then show ?thesis
  using true-clss-clss-left-right[OF N-C-M, of {{#}}] unfolding A by auto
qed
have  $N \models_{pm} \text{image-mset } \text{uminus } ?C + \{\# - ?K\# \}$ 
  unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
  proof (intro allI impI)
    fix I
    assume
      tot: total-over-set I (atms-of-ms (set-mset N  $\cup$  {image-mset uminus ?C + {# - ?K#}})) and
      cons: 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  $\{a \in \text{set } M. \text{is-marked } a \wedge a \neq \text{Marked } K ()\} =$ 
       $\text{set } M \cap \{L. \text{is-marked } L \wedge L \neq \text{Marked } K ()\}$ 
      by auto
    then have
      tI: total-over-set I (atm-of 'lit-of ' (set M  $\cap$  {L. is-marked L  $\wedge$  L  $\neq$  Marked K d}))
      using tot by (auto simp add: L atms-of-uminus-lit-atm-of-lit-of)

then have H:  $\bigwedge x.$ 
   $\text{lit-of } x \notin I \implies x \in \text{set } M \implies \text{is-marked } x$ 
   $\implies x \neq \text{Marked } K d \implies -\text{lit-of } x \in I$ 
  proof -
    fix x :: ('v, unit, unit) marked-lit
    assume a1:  $x \neq \text{Marked } K d$ 
    assume a2: is-marked x
    assume a3:  $x \in \text{set } M$ 
    assume a4:  $\text{lit-of } x \notin I$ 
    have atm-of (lit-of x)  $\in \text{atm-of 'lit-of '}$ 
      (set M  $\cap$  {m. is-marked m  $\wedge$  m  $\neq$  Marked K d})
      using a3 a2 a1 by blast
    then have Pos (atm-of (lit-of x))  $\in I \vee \text{Neg (atm-of (lit-of x))} \in I$ 
      using tI unfolding total-over-set-def by blast
    then show  $-\text{lit-of } x \in I$ 
      using a4 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
        literal.sel(1,2))
  qed
have  $\neg I \models_s ?C'$ 
  using (set-mset N  $\cup$  ?C'  $\models_{ps} \{\{\#\}\}$ ) tot cons (I  $\models_{sm} N$ )
  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  $I \models \text{image-mset } \text{uminus } ?C + \{\# - \text{lit-of } L\# \}$ 
  unfolding true-clss-def true-clss-def
  using  $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$ 
  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  $F \models_{as} CNot (\text{image-mset } \text{uminus } ?C)$ 
    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 $M M' :: ('v, unit, unit) \text{ marked-lit list}$

assumes

backtrack: *backtrack* $S T$ **and**

no-dup: $(no\text{-}dup \circ fst) S$ **and**

decomp: *all-decomposition-implies-m* $(snd S)$ (*get-all-marked-decomposition* $(fst S)$)

shows

$\exists C F' K F L l C'.$

$fst S = F' @ \text{Marked } K () \# F \wedge$

$T = (\text{Propagated } L l \# F, snd S) \wedge C \in \# snd S \wedge fst S \models_{as} CNot C$

$\wedge \text{undefined-lit } F L \wedge atm\text{-of } L \in atm\text{-of-mm } (snd S) \cup atm\text{-of } ' \text{ lits-of-l } (fst S) \wedge$

$snd S \models_{pm} C' + \{\#L\# \} \wedge F \models_{as} CNot C'$

apply (*cases* S , *cases* T)

using *backtrack-is-backjump*[*of* $fst S$ $snd S$ $fst T$ $snd T$] *assms* **by** *fastforce*

sublocale *dpll-state*

id op $\# \cup \lambda L C. C + \{\#L\# \}$ *remove1-mset*

id op $+ op \in \# \lambda L C. C + \{\#L\# \}$ *remove1-mset*

fst snd $\lambda L (M, N). (L \# M, N) \lambda (M, N). (tl M, N)$

$\lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, \text{removeAll-mset } C N)$

by *unfold-locales* (*auto simp: ac-simps*)

sublocale *backjumping-ops*

id op $\# \cup \lambda L C. C + \{\#L\# \}$ *remove1-mset*

id op $+ op \in \# \lambda L C. C + \{\#L\# \}$ *remove1-mset*

fst snd $\lambda L (M, N). (L \# M, N) \lambda (M, N). (tl M, N)$

$\lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, \text{removeAll-mset } C N) \lambda - - S T. \text{backtrack } S T$

by *unfold-locales*

lemma *reduce-trail-to_{NOT}-snd*:

$snd (\text{reduce-trail-to}_{NOT} F S) = snd S$

apply (*induction* $F S$ *rule: reduce-trail-to_{NOT}.induct*)

by (*cases* S , *rename-tac* $F Sa$, *case-tac* Sa)

(*simp add: less-imp-diff-less reduce-trail-to_{NOT}.simps*)

lemma *reduce-trail-to_{NOT}*:

$\text{reduce-trail-to}_{NOT} F S =$

(*if* $\text{length } (fst S) \geq \text{length } F$

then $\text{drop } (\text{length } (fst S) - \text{length } F) (fst S)$

else \square ,

$snd S$) (*is* $?R = ?C$)

proof –

have $?R = (fst ?R, snd ?R)$

by *auto*

also have $(fst ?R, snd ?R) = ?C$

by (*auto simp: trail-reduce-trail-to_{NOT}-drop reduce-trail-to_{NOT}-snd*)

finally show *?thesis* .

qed

lemma *backtrack-is-backjump''*:

fixes $M M' :: ('v, unit, unit) \text{ marked-lit list}$

assumes

backtrack: *backtrack* $S T$ **and**

no-dup: $(no\text{-}dup \circ fst) S$ **and**

$decomp$: $all-decomposition-implies-m$ ($snd\ S$) ($get-all-marked-decomposition$ ($fst\ S$))
shows $backjump\ S\ T$
proof –
obtain $C\ F'\ K\ F\ L\ l\ C'$ **where**
1: $fst\ S = F' @ Marked\ K\ () \# F$ **and**
2: $T = (Propagated\ L\ l \# F, snd\ S)$ **and**
3: $C \in \#\ snd\ S$ **and**
4: $fst\ S \models_{as} CNot\ C$ **and**
5: $undefined-lit\ F\ L$ **and**
6: $atm-of\ L \in atm-of-mm\ (snd\ S) \cup atm-of\ 'lits-of-l\ (fst\ S)$ **and**
7: $snd\ S \models_{pm} C' + \{\#L\# \}$ **and**
8: $F \models_{as} CNot\ C'$
using $backtrack-is-backjump'[OF\ assms]$ **by** $force$
show $?thesis$
apply ($cases\ S$)
using $backjump.intros[OF\ 1\ -\ -\ 4\ 5\ -\ -\ 8, of\ T]\ 2\ backtrack\ 1\ 5\ 3\ 6\ 7$
by ($auto\ simp$: $state-eq_{NOT}$ - $def\ trail-reduce-trail-to_{NOT}$ - $drop$
 $reduce-trail-to_{NOT}\ simp\ del$: $state-simp_{NOT}$)
qed

lemma $can-do-bt-step$:
assumes
 M : $fst\ S = F' @ Marked\ K\ d \# F$ **and**
 $C \in \#\ snd\ S$ **and**
 C : $fst\ S \models_{as} CNot\ C$
shows $\neg no-step\ backtrack\ S$
proof –
obtain $L\ G'\ G$ **where**
 $backtrack-split\ (fst\ S) = (G', L \# G)$
unfolding M **by** ($induction\ F'$ $rule$: $marked-lit-list-induct$) $auto$
moreover then have $is-marked\ L$
by ($metis\ backtrack-split-snd-hd-marked\ list.distinct(1)\ list.sel(1)\ snd-conv$)
ultimately show $?thesis$
using $backtrack.intros[of\ S\ G'\ L\ G\ C]\ (C \in \#\ snd\ S)\ C$ **unfolding** M **by** $auto$
qed

end

sublocale $dpll-with-backtrack \subseteq dpll-with-backjumping-ops$
 $id\ op\ \# \cup\ \lambda L\ C. C + \{\#L\#\}\ remove1-mset$
 $id\ op + op \in \# \lambda L\ C. C + \{\#L\#\}\ remove1-mset$
 $fst\ snd\ \lambda L\ (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C\ (M, N). (M, \{\#C\#\} + N) \lambda C\ (M, N). (M, removeAll-mset\ C\ N)$
 $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-marked-decomposition\ M)$
 $\lambda- -\ S\ T. backtrack\ S\ T$
 $\lambda- -. True$
apply $unfold-locales$
by ($metis\ (mono-tags, lifting)\ case-prod-beta\ comp-def\ dpll-with-backtrack.backtrack-is-backjump''$
 $dpll-with-backtrack.can-do-bt-step\ id-apply$)

sublocale $dpll-with-backtrack \subseteq dpll-with-backjumping$
 $id\ op\ \# \cup\ \lambda L\ C. C + \{\#L\#\}\ remove1-mset$
 $id\ op + op \in \# \lambda L\ C. C + \{\#L\#\}\ remove1-mset$
 $fst\ snd\ \lambda L\ (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C\ (M, N). (M, \{\#C\#\} + N) \lambda C\ (M, N). (M, removeAll-mset\ C\ N)$

```

λ(M, N). no-dup M ∧ all-decomposition-implies-m N (get-all-marked-decomposition M)
λ- - S T. backtrack S T
λ- -. True
apply unfold-locales
using dpll-bj-no-dup dpll-bj-all-decomposition-implies-inv apply fastforce
done

context dpll-with-backtrack
begin
term learn
end

context dpll-with-backtrack
begin
lemma wf-tranclp-dpll-initail-state:
  assumes fin: finite A
  shows wf {((M':('v, unit, unit) marked-lits, N':('v clauses), ([], N))|M' N' N.
    dpll-bj++ ([], N) (M', N') ∧ atms-of-mm 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 distinct-consistent-interp satisfiable-carac' true-annots-true-cls by blast
  then show ?thesis
  using full-dpll-final-state-conclusive[OF full] by auto
qed

interpretation conflict-driven-clause-learning-ops
  id op #∪ λL C. C + {#L#} remove1-mset
  id op + op ∈# λL C. C + {#L#} remove1-mset
  fst snd λL (M, N). (L # M, N)
  λ(M, N). (tl M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, removeAll-mset C N)
  λ(M, N). no-dup M ∧ all-decomposition-implies-m N (get-all-marked-decomposition M)
  λ- - S T. backtrack S T
  λ- -. True λ- -. False λ- -. False
by unfold-locales

interpretation conflict-driven-clause-learning
  id op #∪ λL C. C + {#L#} remove1-mset

```

```

  id op + op ∈ # λL C. C + {#L#} remove1-mset
  fst snd λL (M, N). (L # M, N)
λ(M, N). (tl M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, removeAll-mset C N)
λ(M, N). no-dup M ∧ all-decomposition-implies-m N (get-all-marked-decomposition M)
λ- - S T. backtrack S T
λ- -. True λ- -. False λ- -. False
apply unfold-locales
using cdclNOT-all-decomposition-implies cdclNOT-no-dup by fastforce

```

lemma *cdcl_{NOT}-is-dpll*:
cdcl_{NOT} S T \longleftrightarrow *dpll-bj S T*
by (auto simp: *cdcl_{NOT}.simps learn.simps forget_{NOT}.simps*)

Another proof of termination:

lemma *wf {(T, S). dpll-bj S T ∧ cdcl_{NOT}-NOT-all-inv A S}*
unfolding *cdcl_{NOT}-is-dpll[symmetric]*
by (rule *wf-cdcl_{NOT}-no-learn-and-forget-infinite-chain*)
(auto simp: *learn.simps forget_{NOT}.simps*)
end

16.2 Adding restarts

```

locale dpll-withbacktrack-and-restarts =
  dpll-with-backtrack +
  fixes f :: nat ⇒ nat
  assumes unbounded: unbounded f and f-ge-1:  $\bigwedge n. n \geq 1 \implies f n \geq 1$ 
begin
  sublocale cdclNOT-increasing-restarts
  id op # ∪ λL C. C + {#L#} remove1-mset
  id op + op ∈ # λL C. C + {#L#} remove1-mset
  fst snd λL (M, N). (L # M, N) λ(M, N). (tl M, N)
  λC (M, N). (M, {#C#} + N) λC (M, N). (M, removeAll-mset C N) f λ(-, N) S. S = ([], N)
  λA (M, N). atms-of-mm N ⊆ atms-of-ms A ∧ atm-of ‘ lits-of-l M ⊆ atms-of-ms A ∧ finite A
  ∧ all-decomposition-implies-m N (get-all-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 id-apply prod.case-eq-if)
  apply (rule dpll-bj-trail-mes-decreasing-prop; auto)
  apply (rename-tac A T U, case-tac T, simp)
  apply (rename-tac A T U, 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

```


17 DPLL

17.1 Rules

type-synonym $'a \text{ dpll}_W\text{-marked-lit} = ('a, \text{unit}, \text{unit}) \text{ marked-lit}$
type-synonym $'a \text{ dpll}_W\text{-marked-lits} = ('a, \text{unit}, \text{unit}) \text{ marked-lits}$
type-synonym $'v \text{ dpll}_W\text{-state} = 'v \text{ dpll}_W\text{-marked-lits} \times 'v \text{ clauses}$

abbreviation $\text{trail} :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ dpll}_W\text{-marked-lits}$ **where**
 $\text{trail} \equiv \text{fst}$
abbreviation $\text{clauses} :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ clauses}$ **where**
 $\text{clauses} \equiv \text{snd}$

The definition of DPLL is given in figure 2.13 page 70 of CW.

inductive $\text{dpll}_W :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ dpll}_W\text{-state} \Rightarrow \text{bool}$ **where**
 $\text{propagate: } C + \{\#L\# \} \in \# \text{ clauses } S \Longrightarrow \text{trail } S \models_{\text{as}} C \text{Not } C \Longrightarrow \text{undefined-lit } (\text{trail } S) \ L$
 $\Longrightarrow \text{dpll}_W \ S \ (\text{Propagated } L \ () \ \# \ \text{trail } S, \text{ clauses } S) \mid$
 $\text{decided: } \text{undefined-lit } (\text{trail } S) \ L \Longrightarrow \text{atm-of } L \in \text{atms-of-mm } (\text{clauses } S)$
 $\Longrightarrow \text{dpll}_W \ S \ (\text{Marked } L \ () \ \# \ \text{trail } S, \text{ clauses } S) \mid$
 $\text{backtrack: } \text{backtrack-split } (\text{trail } S) = (M', L \# M) \Longrightarrow \text{is-marked } L \Longrightarrow D \in \# \text{ clauses } S$
 $\Longrightarrow \text{trail } S \models_{\text{as}} C \text{Not } D \Longrightarrow \text{dpll}_W \ S \ (\text{Propagated } (- \ (\text{lit-of } L)) \ () \ \# \ M, \text{ clauses } S)$

17.2 Invariants

lemma $\text{dpll}_W\text{-distinct-inv:}$
assumes $\text{dpll}_W \ S \ S'$
and $\text{no-dup } (\text{trail } S)$
shows $\text{no-dup } (\text{trail } S')$
using assms
proof ($\text{induct rule: } \text{dpll}_W.\text{induct}$)
case ($\text{decided } L \ S$)
then show $?case$ **using** $\text{defined-lit-map by force}$
next
case ($\text{propagate } C \ L \ S$)
then show $?case$ **using** $\text{defined-lit-map by force}$
next
case ($\text{backtrack } S \ M' \ L \ M \ D$) **note** $\text{extracted} = \text{this}(1)$ **and** $\text{no-dup} = \text{this}(5)$
show $?case$
using $\text{no-dup backtrack-split-list-eq[of trail } S, \text{ symmetric}]$ **unfolding** extracted by auto
qed

lemma $\text{dpll}_W\text{-consistent-interp-inv:}$
assumes $\text{dpll}_W \ S \ S'$
and $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$
and $\text{no-dup } (\text{trail } S)$
shows $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S'))$
using assms
proof ($\text{induct rule: } \text{dpll}_W.\text{induct}$)
case ($\text{backtrack } S \ M' \ L \ M \ D$) **note** $\text{extracted} = \text{this}(1)$ **and** $\text{marked} = \text{this}(2)$ **and** $D = \text{this}(4)$ **and**
 $\text{cons} = \text{this}(5)$ **and** $\text{no-dup} = \text{this}(6)$
have $\text{no-dup}' : \text{no-dup } M$
by ($\text{metis } (\text{no-types}) \text{backtrack-split-list-eq distinct.simps}(2) \text{distinct-append extracted}$
 $\text{list.simps}(9) \text{map-append no-dup snd-conv}$)
then have $\text{insert } (\text{lit-of } L) \ (\text{lits-of-l } M) \subseteq \text{lits-of-l } (\text{trail } S)$
using $\text{backtrack-split-list-eq[of trail } S, \text{ symmetric}]$ **unfolding** extracted by auto
then have $\text{cons: consistent-interp } (\text{insert } (\text{lit-of } L) \ (\text{lits-of-l } M))$

using *consistent-interp-subset* cons by blast
 moreover
 have *lit-of* $L \notin \text{ lits-of-l } M$
 using *no-dup backtrack-split-list-eq*[of trail S , *symmetric*] *extracted*
 unfolding *lits-of-def* by force
 moreover
 have *atm-of* $(\neg \text{ lit-of } L) \notin (\lambda m. \text{ atm-of } (\text{ lit-of } m)) \text{ ' set } M$
 using *no-dup backtrack-split-list-eq*[of trail S , *symmetric*] **unfolding** *extracted* by force
 then have $\neg \text{ lit-of } L \notin \text{ lits-of-l } M$
 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* ' $(\text{ lits-of-l } (\text{ trail } S)) \subseteq \text{ atms-of-mm } (\text{ clauses } S)$
 shows *atm-of* ' $(\text{ lits-of-l } (\text{ trail } S')) \subseteq \text{ atms-of-mm } (\text{ clauses } S')$
 using *assms*
proof (*induct* rule: *dpll_W.induct*)
 case (*backtrack* $S M' L M D$)
 then have *atm-of* $(\text{ lit-of } L) \in \text{ atms-of-mm } (\text{ clauses } S)$
 using *backtrack-split-list-eq*[of trail S , *symmetric*] by auto
 moreover
 have *atm-of* ' $\text{ lits-of-l } (\text{ trail } S) \subseteq \text{ atms-of-mm } (\text{ clauses } S)$
 using *backtrack*(5) by simp
 then have $\bigwedge x. x \in \text{ set } M \implies \text{ atm-of } (\text{ lit-of } x) \in \text{ atms-of-mm } (\text{ clauses } S)$
 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* $((\lambda a. \{\# \text{ lit-of } a \# \}) \text{ ' } c) = \text{ atm-of ' lit-of ' } c$
 unfolding *atms-of-ms-def* using *image-iff* by force

Lemma theorem 2.8.2 page 71 of CW

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* ' $\text{ lits-of-l } (\text{ trail } S) \subseteq \text{ atms-of-mm } (\text{ clauses } S)$
 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* $(\lambda a. \{\# \text{ lit-of } a \# \}) (\text{ trail } S) \cup \text{ set-mset } (\text{ clauses } S)$)
 have ?*I* $\models_p C + \{\# L \# \}$ by (auto simp add: *inS*)
 moreover have ?*I* $\models_{ps} C \text{ Not } C$ using *true-annots-true-clss-cls cnot* by *fastforce*
 ultimately have ?*I* $\models_p \{\# L \# \}$ 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  $S$ )  $\neq []$ 
  have 1:  $\bigwedge a\ b. (a, b) \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } (\text{trail } S)))$ 
     $\implies (\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S)) \models_{ps} \text{unmark-l } b$ 
    using IH unfolding all-decomposition-implies-def by (fastforce simp add: list.set-sel(2) n)
  moreover have 2:  $\bigwedge a\ c. \text{hd } (\text{get-all-marked-decomposition } (\text{trail } S)) = (a, c)$ 
     $\implies (\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S)) \models_{ps} (\text{unmark-l } c)$ 
    by (metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single list.collapse n)
  moreover have 3:  $\bigwedge a\ c. \text{hd } (\text{get-all-marked-decomposition } (\text{trail } S)) = (a, c)$ 
     $\implies (\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S)) \models_p \{\#L\#$ 
  proof –
    fix  $a\ c$ 
    assume  $h$ :  $\text{hd } (\text{get-all-marked-decomposition } (\text{trail } S)) = (a, c)$ 
    have  $h'$ :  $\text{trail } S = c @ a$  using get-all-marked-decomposition-decomp  $h$  by blast
    have  $I$ :  $\text{set } (\text{map } (\lambda a. \{\#lit\text{-of } a\#\})\ a) \cup \text{set-mset } (\text{clauses } S)$ 
       $\cup \text{unmark-l } c \models_{ps} \text{CNot } C$ 
    using  $\langle ?I \models_{ps} \text{CNot } C \rangle$  unfolding  $h'$  by (simp add: Un-commute Un-left-commute)
    have
       $\text{atms-of-ms } (\text{CNot } C) \subseteq \text{atms-of-ms } (\text{set } (\text{map } (\lambda a. \{\#lit\text{-of } a\#\})\ a) \cup \text{set-mset } (\text{clauses } S))$ 
      and
       $\text{atms-of-ms } (\text{unmark-l } c) \subseteq \text{atms-of-ms } (\text{set } (\text{map } (\lambda a. \{\#lit\text{-of } a\#\})\ a) \cup \text{set-mset } (\text{clauses } S))$ 
    apply (metis CNot-plus Un-subset-iff atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of atms-of-ms-union inS sup.coboundedI2)
    using inS atms-of-atms-of-ms-mono atms-incl by (fastforce simp: h')

    then have  $\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S) \models_{ps} \text{CNot } C$ 
    using true-clss-clss-left-right[OF - I] h 2 by auto
    then show  $\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S) \models_p \{\#L\#$ 
    by (metis (no-types) Un-insert-right inS insertI1 mk-disjoint-insert inS true-clss-clss-in true-clss-clss-plus-CNot)
    qed
  ultimately have  $?case$ 
    by (cases hd (get-all-marked-decomposition (trail S)))
    (auto simp: all-decomposition-implies-def)
}
ultimately show  $?case$  by auto
next
case (backtrack  $S\ M'\ L\ M\ D$ ) note  $\text{extracted} = \text{this}(1)$  and  $\text{marked} = \text{this}(2)$  and  $D = \text{this}(3)$  and
 $\text{cnot} = \text{this}(4)$  and  $\text{cons} = \text{this}(4)$  and  $IH = \text{this}(5)$  and  $\text{atms-incl} = \text{this}(6)$ 
have  $S$ :  $\text{trail } S = M' @ L \# M$ 
using backtrack-split-list-eq[of trail S] unfolding  $\text{extracted}$  by auto
have  $M'$ :  $\forall l \in \text{set } M'. \neg \text{is-marked } l$ 
using  $\text{extracted}$  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  $S$ )  $((L \# M, M')$ 
   $\# \text{tl } (\text{get-all-marked-decomposition } (\text{trail } S)))$ 
by (metis (no-types) IH extracted get-all-marked-decomposition-backtrack-split list.exhaust-sel)
then have 1:  $\text{unmark-l } (L \# M) \cup \text{set-mset } (\text{clauses } S) \models_{ps} (\lambda a. \{\#lit\text{-of } a\#\})\ \text{'set } M'$ 
by simp
moreover
have  $\text{unmark-l } (L \# M) \cup \text{unmark-l } M' \models_{ps} \text{CNot } D$ 
by (metis (mono-tags, lifting) S Un-commute cons image-Un set-append true-annots-true-clss-clss)

```

```

then have 2: unmark-l ( $L \# M$ )  $\cup$  set-mset (clauses  $S$ )  $\cup$  unmark-l  $M'$ 
   $\models_{ps}$  CNot  $D$ 
by (metis (no-types, lifting) Un-assoc Un-left-commute true-clss-clss-union-l-r)
ultimately
have set (map ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ( $L \# M$ ))  $\cup$  set-mset (clauses  $S$ )  $\models_{ps}$  CNot  $D$ 
  using true-clss-clss-left-right by fastforce
then have set (map ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ( $L \# M$ ))  $\cup$  set-mset (clauses  $S$ )  $\models_p$   $\{\#\}$ 
  by (metis (mono-tags, lifting)  $D$  Un-def mem-Collect-eq
    true-clss-clss-contradiction-true-clss-clss-false)
then have  $IL$ : unmark-l  $M \cup$  set-mset (clauses  $S$ )  $\models_p$   $\{\#-lit\text{-of } L\#\}$ 
  using true-clss-clss-false-left-right by auto
show ?case unfolding  $S$  all-decomposition-implies-def
proof
  fix  $x$   $P$  level
  assume  $x$ :  $x \in$  set (get-all-marked-decomposition
    (fst (Propagated ( $- lit\text{-of } L$ )  $P \# M$ , clauses  $S$ )))
  let ? $M'$  = Propagated ( $- lit\text{-of } L$ )  $P \# M$ 
  let ? $hd$  = hd (get-all-marked-decomposition ? $M'$ )
  let ? $tl$  = tl (get-all-marked-decomposition ? $M'$ )
  have  $x = ?hd \vee x \in$  set ? $tl$ 
    using  $x$ 
    by (cases get-all-marked-decomposition ? $M'$ )
      auto
  moreover {
    assume  $x'$ :  $x \in$  set ? $tl$ 
    have  $L'$ : Marked (lit-of  $L$ ) () =  $L$  using marked by (cases  $L$ , auto)
    have  $x \in$  set (get-all-marked-decomposition ( $M' @ L \# M$ ))
      using  $x'$  get-all-marked-decomposition-except-last-choice-equal[of  $M' lit\text{-of } L P M$ ]
       $L'$  by (metis (no-types)  $M'$  list.set-sel(2) tl-Nil)
    then have case  $x$  of ( $Ls$ , seen)  $\Rightarrow$  unmark-l  $Ls \cup$  set-mset (clauses  $S$ )
       $\models_{ps}$  unmark-l seen
      using marked IH by (cases  $L$ ) (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  $f1$ :  $\bigwedge ms. \text{length} (\text{get-all-marked-decomposition } (M' @ ms))$ 
        =  $\text{length} (\text{get-all-marked-decomposition } ms)$ 
      by (simp add: M' get-all-marked-decomposition-remove-unmark-ssed-length)
      have  $Suc (\text{length} (\text{get-all-marked-decomposition } M)) \neq Suc 0$ 
      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  $x''$ :  $x = (M0, \text{Propagated } (-lit\text{-of } L) P \# M0')$ 
    unfolding  $x'$  using get-all-marked-decomposition-last-choice  $tl M' L0$ 
    by (metis marked marked-lit.collapse(1))
  obtain  $l\text{-get-all-marked-decomposition}$  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' : *unmark-l* $M0 \cup \text{set-mset}$ (*clauses S*)
 $\cup \text{unmark-l } M0' \models_{ps} \{\{\# - \text{lit-of } L\#\}\}$
using IL **by** (*simp add: Un-commute Un-left-commute image-Un*)
moreover have H : *unmark-l* $M0 \cup \text{set-mset}$ (*clauses S*)
 $\models_{ps} \text{unmark-l } M0'$
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 unmark-l Ls \cup set-mset* (*clauses S*)
 $\models_{ps} \text{unmark-l seen}$
using *true-clss-clss-left-right* **unfolding** x'' **by** *auto*
}
ultimately show *case x of (Ls, seen) \Rightarrow*
unmark-l Ls \cup set-mset (snd (?M', clauses S))
 $\models_{ps} \text{unmark-l seen}$
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-l* (*trail S*) \subseteq *atms-of-mm* (*clauses S*)
shows *set-mset* (*clauses S'*) $\cup \{\{\#\text{lit-of } L\#\} \mid L. \text{is-marked } L \wedge L \in \text{set} (\text{trail } S')\}$
 $\models_{ps} (\lambda a. \{\#\text{lit-of } a\#\}) ' \bigcup (\text{set ' snd ' set } (\text{get-all-marked-decomposition } (\text{trail } S')))$
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-l* $M \subseteq \text{atms-of-ms } N$
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 $l0$: $\{\{\#\text{lit-of } L\#\} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} = \{\}$ **using** *marked* **by** *auto*
have *atms-of-ms* ($N \cup \text{unmark-l } M$) = *atms-of-ms* N
using *atm-incl* **unfolding** *atms-of-ms-def lits-of-def* **by** *auto*

then have *total-over-m* I ($N \cup (\lambda a. \{\#\text{lit-of } a\#\}) ' (\text{set } M)$)
using *tot* **unfolding** *total-over-m-def* **by** *auto*
then have $I \models_s (\lambda a. \{\#\text{lit-of } a\#\}) ' (\text{set } M)$
using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons I*

```

  unfolding true-clss-clss-def l0 by auto
then have IM:  $I \models_s \text{unmark-l } M$  by auto
{
  fix K
  assume  $K \in \# D$ 
  then have  $-K \in \text{lits-of-l } M$ 
    by (auto split: if-split-asm
      intro: allE[OF D[unfolded true-annots-def Ball-def], of { $\#-K$ }])
  then have  $-K \in I$  using IM true-clss-singleton-lit-of-implies-incl by fastforce
}
then have  $\neg I \models D$  using cons unfolding true-clss-def consistent-interp-def by auto
then show False using I-D by blast
qed

```

lemma *dpll_W-same-clauses*:

```

  assumes dpllW S S'
  shows clauses S = clauses S'
  using assms by (induct rule: dpllW.induct, auto)

```

lemma *rtrancpl-dpll_W-inv*:

```

  assumes rtrancpl dpllW S S'
  and inv: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
  and atm-incl: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-mm (clauses S)
  and consistent-interp (lits-of-l (trail S))
  and no-dup (trail S)
  shows all-decomposition-implies-m (clauses S') (get-all-marked-decomposition (trail S'))
  and atm-of ' lits-of-l (trail S')  $\subseteq$  atms-of-mm (clauses S')
  and clauses S = clauses S'
  and consistent-interp (lits-of-l (trail S'))
  and no-dup (trail S')
  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-l (trail S)  $\subseteq$  atms-of-mm (clauses S) and
  clauses S = clauses S and
  consistent-interp (lits-of-l (trail S)) and
  no-dup (trail S) 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-l (trail S)  $\subseteq$  atms-of-mm (clauses S) and
  cons: consistent-interp (lits-of-l (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-l (trail S')  $\subseteq$  atms-of-mm (clauses S') and

```

```

  snd: clauses S = clauses S' and

```

```

  cons': consistent-interp (lits-of-l (trail S')) and

```

```

  no-dup': no-dup (trail S') by blast+

```

show clauses S = clauses S'' using dpll_W-same-clauses[OF dpll_W] 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-l (trail S'')  $\subseteq$  atms-of-mm (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-l (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-l (trail S)  $\subseteq$  atms-of-mm (clauses S)
   $\wedge$  consistent-interp (lits-of-l (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-l (trail S)  $\subseteq$  atms-of-mm (clauses S)
  and consistent-interp (lits-of-l (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 dpllW-all-inv S'
  using assms rtranclp-dpllW-all-inv by blast

```

```

lemma rtranclp-dpllW-inv-starting-from-0:
  assumes rtranclp dpllW S S'
  and inv: trail S = []
  shows dpllW-all-inv S'
proof –
  have dpllW-all-inv S
    using assms unfolding all-decomposition-implies-def dpllW-all-inv-def by auto
  then show ?thesis using rtranclp-dpllW-all-inv[OF assms(1)] by blast
qed

```

```

lemma dpllW-can-do-step:
  assumes consistent-interp (set M)
  and distinct M
  and atm-of ' (set M)  $\subseteq$  atms-of-mm N
  shows rtranclp dpllW ([], 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-lit (map (λM. Marked M ()) M) L
  unfolding defined-lit-def consistent-interp-def by auto
moreover have atm-of L ∈ atms-of-mm N using Cons.premis(3) by auto
ultimately have dpllW (map (λM. Marked M ()) M, N) (map (λM. Marked M ()) (L # M), N)
  using dpllW.decided by auto
moreover have consistent-interp (set M) and distinct M and atm-of ‘ set M ⊆ atms-of-mm N
  using Cons.premis unfolding consistent-interp-def by auto
ultimately show ?case using Cons.hyps by auto
qed

```

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

lemma *dpll_W-strong-completeness*:

```

assumes set M ⊢sm N
and consistent-interp (set M)
and distinct M
and atm-of ‘ (set M) ⊆ atms-of-mm N
shows dpllW** ([], N) (map (λM. Marked M ()) M, N)
and conclusive-dpllW-state (map (λM. Marked M ()) M, N)
proof –
show rtrancpl dpllW ([], N) (map (λM. Marked M ()) M, N) using dpllW-can-do-step assms by auto
have map (λM. Marked M ()) M ⊢asm N using assms(1) true-annots-marked-true-cl by auto
then show conclusive-dpllW-state (map (λM. Marked M ()) M, N)
  unfolding conclusive-dpllW-state-def by auto
qed

```

lemma *dpll_W-sound*:

```

assumes
  rtrancpl dpllW ([], N) (M, N) and
  ∀ S. ¬dpllW (M, N) S
shows M ⊢asm N ⟷ satisfiable (set-mset N) (is ?A ⟷ ?B)
proof
let ?M' = lits-of-l M
assume ?A
then have ?M' ⊢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: ¬ ?A
have (∃ L. undefined-lit M L ∧ atm-of L ∈ atms-of-mm N) ∨ (∃ D ∈ #N. M ⊢as CNot D)
proof –
obtain D :: 'a clause where D: D ∈ # N and ¬ M ⊢a D
  using n unfolding true-annots-def Ball-def by auto
then have (∃ L. undefined-lit M L ∧ atm-of L ∈ atms-of D) ∨ M ⊢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
    by (metis Bex-def D atms-of-atms-of-ms-mono rev-subsetD)
  qed
  moreover {
    assume  $\exists L. \text{undefined-lit } M L \wedge \text{atm-of } L \in \text{atms-of-mm } N$ 
    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  $\forall l \in \text{set } M. \neg \text{is-marked } l$ 
      moreover have dpllW-all-inv ([], N)
      using assms unfolding all-decomposition-implies-def dpllW-all-inv-def by auto
      ultimately have unsatisfiable (set-mset N)
      using only-propagated-vars-unsat[of M D set-mset N] DN MD
      rtranclp-dpllW-all-inv[OF assms(1)] by force
      then have False using <?B> by blast
    }
    moreover {
      assume  $l: \exists l \in \text{set } M. \text{is-marked } l$ 
      then have False
      using backtrack[of (M, N) - - D] DN MD assms(2)
      backtrack-split-some-is-marked-then-snd-has-hd[OF l]
      by (metis backtrack-split-snd-hd-marked fst-conv list.distinct(1) list.sel(1) snd-conv)
    }
    ultimately have False by blast
  }
  ultimately show False by blast
qed
qed

```

17.3 Termination

definition $dpll_W\text{-mes } M n =$
 $\text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } (1::nat)) (\text{rev } M) @ \text{replicate } (n - \text{length } M) 3$

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** $dpll_W\text{-mes-def}$ **by** *auto*

lemma $\text{distinctcard-atm-of-lit-of-eq-length}$:
assumes $\text{no-dup } S$
shows $\text{card } (\text{atm-of } \text{'lits-of-l } S) = \text{length } S$
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 $m: \text{map } (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$

```

    @ replicate (card vars - length (trail S)) 3
  = map (λl. if is-marked l then 2 else 1) (rev (trail S)) @ 3
    # replicate (card vars - Suc (length (trail S))) 3
  using propagate.premis[simplified] using Suc-diff-le by fastforce
then show ?case
  using propagate.premis(1) unfolding dpllW-mes-def by (fastforce simp add: lexn-conv assms(2))
next
case (decided S L)
have m: map (λl. if is-marked l then 2 else 1) (rev (trail S))
  @ replicate (card vars - length (trail S)) 3
  = map (λl. if is-marked l then 2 else 1) (rev (trail S)) @ 3
  # replicate (card vars - Suc (length (trail S))) 3
  using decided.premis[simplified] using Suc-diff-le by fastforce
then show ?case
  using decided.premis unfolding dpllW-mes-def by (force simp add: lexn-conv assms(2))
next
case (backtrack S M' L M D)
have L: is-marked L using backtrack.hyps(2) by auto
have S: trail S = M' @ L # M
  using backtrack.hyps(1) backtrack-split-list-eq[of trail S] by auto
show ?case
  using backtrack.premis L unfolding dpllW-mes-def S by (fastforce simp add: lexn-conv assms(2))
qed

```

Proposition theorem 2.8.7 page 73 of CW

lemma dpll_W-card-decrease':

```

  assumes dpll: dpllW S S'
  and atm-incl: atm-of ' lits-of-l (trail S) ⊆ atms-of-mm (clauses S)
  and no-dup: no-dup (trail S)
  shows (dpllW-mes (trail S') (card (atms-of-mm (clauses S'))),
        dpllW-mes (trail S) (card (atms-of-mm (clauses S)))) ∈ lex {(a, b). a < b}

```

proof –

```

  have finite (atms-of-mm (clauses S)) unfolding atms-of-ms-def by auto
  then have 1: length (trail S) ≤ card (atms-of-mm (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-l (trail S') ⊆ atms-of-mm (clauses S')
    using atm-incl dpll dpllW-vars-in-snd-inv[OF dpll] by force
  have finite (atms-of-mm (clauses S'))
    unfolding atms-of-ms-def by auto
  then have 2: length (trail S') ≤ card (atms-of-mm (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-mm (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-mm (clauses S))))
  ∈ lexn {(a, b). a < b} (card (atms-of-mm (clauses S)))
  using dpllW-card-decrease[OF assms(1), of atms-of-mm (clauses S)] by blast
then have (dpllW-mes (trail S') (card (atms-of-mm (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-mm (clauses S)))) ∈ lex {(a, b). a < b}
  unfolding lex-def by auto
then show (dpllW-mes (trail S') (card (atms-of-mm (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-mm (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-mm (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-mm (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-trancpl')
      next
        case (trancpl-into-trancpl S S' S'')
        then have (S', S) ∈ {a. case a of (S', S) ⇒ dpllW-all-inv S ∧ dpllW S S'}+ by blast
        moreover have dpllW-all-inv S'
          using rtrancpl-dpllW-all-inv[OF trancpl-into-rtrancpl[OF trancpl-into-trancpl.hyps(1)]]
          trancpl-into-trancpl.prem by auto
        ultimately have (S'', S') ∈ {(pa, p). dpllW-all-inv p ∧ dpllW p pa}+
          using (dpllW-all-inv S') trancpl-into-trancpl.hyps(3) by blast
        then show ?case
          using (S', S) ∈ {a. case a of (S', S) ⇒ dpllW-all-inv S ∧ dpllW S S'}+ by auto
      qed
  }
  then show ?B ⊆ ?A by blast
qed

lemma dpllW-wf-trancpl: wf {(S', S). dpllW-all-inv S ∧ dpllW++ S S'}
  unfolding dpllW-trancpl-star-commute[symmetric] by (simp add: dpllW-wf wf-trancpl)

lemma dpllW-wf-plus:
  shows wf {(S', ([], N)) | S'. dpllW++ ([], N) S'} (is wf ?P)
  apply (rule wf-subset[OF dpllW-wf-trancpl, of ?P])
  using assms unfolding dpllW-all-inv-def by auto

```

17.4 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*: $\forall s \in \text{atms-of-mm} (\text{clauses } S). s \in \text{atm-of } \text{'lits-of-l'} (\text{trail } S)$

proof (*rule ccontr*)

assume $\neg (\forall s \in \text{atms-of-mm} (\text{clauses } S). s \in \text{atm-of } \text{'lits-of-l'} (\text{trail } S))$

then obtain L **where**

$L\text{-in-atms}$: $L \in \text{atms-of-mm} (\text{clauses } S)$ **and**

$L\text{-notin-trail}$: $L \notin \text{atm-of } \text{'lits-of-l'} (\text{trail } S)$ **by** *metis*

obtain L' **where** L' : $\text{atm-of } L' = L$ **by** (*meson literal.sel(2)*)

then have *undefined-lit* ($\text{trail } S$) L'

unfolding *Marked-Propagated-in-iff-in-lits-of-l* **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 : *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 $\forall s \in \text{atms-of-ms } \{D\}. s \in \text{atm-of } \text{'lits-of-l'} (\text{trail } S)$

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

then have $\text{trail } S \models_{\text{as}} C \text{Not } D$

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)* *dpll_W.backtrack* $L \langle D \in \# \text{clauses } S \rangle \langle \text{trail } S \models_{\text{as}} C \text{Not } D \rangle$ **by** *blast*

}

moreover {

assume *tr*: $\forall C \in \# \text{clauses } S. \neg \text{trail } S \models_{\text{as}} C \text{Not } C$

obtain C **where** $C\text{-in-cl}$: $C \in \# \text{clauses } S$ **and** $\text{tr}C$: $\neg \text{trail } S \models_a C$

using $\langle \neg \text{trail } S \models_{\text{asm}} \text{clauses } S \rangle$ **unfolding** *true-annots-def* **by** *auto*

have $\forall s \in \text{atms-of-ms } \{C\}. s \in \text{atm-of } \text{'lits-of-l'} (\text{trail } S)$

using *vars* $\langle C \in \# \text{clauses } S \rangle$ **unfolding** *atms-of-ms-def* **by** *auto*

then have $\text{trail } S \models_{\text{as}} C \text{Not } C$

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 *dpll_W-conclusive-state-correct*:

assumes *dpll_W*** ($[], N$) (M, N) **and** *conclusive-dpll_W-state* (M, N)

shows $M \models_{\text{asm}} N \longleftrightarrow \text{satisfiable } (\text{set-mset } N)$ (**is** $?A \longleftrightarrow ?B$)

```

proof
  let ?M' = lits-of-l M
  assume ?A
  then have ?M'  $\models_{sm}$  N by (simp add: true-annots-true-cls)
  moreover have consistent-interp ?M'
    using rtranclp-dpllW-inv-starting-from-0[OF assms(1)] by auto
  ultimately show ?B by auto
next
  assume ?B
  show ?A
  proof (rule ccontr)
    assume n:  $\neg$  ?A
    have no-mark:  $\forall L \in \text{set } M. \neg \text{is-marked } L \ \exists C \in \# N. M \models_{as} C \text{Not } C$ 
      using n assms(2) unfolding conclusive-dpllW-state-def by auto
    moreover obtain D where DN:  $D \in \# N$  and MD:  $M \models_{as} C \text{Not } D$  using no-mark by auto
    ultimately have unsatisfiable (set-mset N)
      using only-propagated-vars-unsat rtranclp-dpllW-all-inv[OF assms(1)]
      unfolding dpllW-all-inv-def by force
    then show False using ⟨?B⟩ by blast
  qed
qed

```

17.5 Link with NOT's DPLL

interpretation *dpll_W-NOT*: *dpll-with-backtrack* .

```

declare dpllW-NOT.state-simpNOT[simp del]
lemma state-eqNOT-iff-eq[iff, simp]: dpllW-NOT.state-eqNOT S T  $\longleftrightarrow$  S = T
  unfolding dpllW-NOT.state-eqNOT-def by (cases S, cases T) auto
lemma dpllW-dpllW-bj:
  assumes inv: dpllW-all-inv S and dpll: dpllW S T
  shows dpllW-NOT.dpll-bj S T
  using dpll inv
  apply (induction rule: dpllW.induct)
    apply (rule dpllW-NOT.bj-propagateNOT)
    apply (rule dpllW-NOT.propagateNOT.propagateNOT; simp?)
    apply fastforce
    apply (rule dpllW-NOT.bj-decideNOT)
    apply (rule dpllW-NOT.decideNOT.decideNOT; simp?)
    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.decideNOTE, cases S)
    apply (frule decided; simp)

  apply (elim dpllW-NOT.propagateNOTE, cases S)
  apply (auto intro!: propagate[of - - (-, -), simplified])[]

```

```

apply (elim  $dpll_{W-NOT}.backjumpE$ , cases  $S$ )
by (simp add:  $dpll_W.simps$   $dpll\_with\_backtrack.backtrack.simps$ )

lemma  $rtrancp\_dpll_W-rtrancp\_dpll_{W-NOT}$ :
  assumes  $dpll_W^{**} S T$  and  $dpll_W-all-inv S$ 
  shows  $dpll_{W-NOT}.dpll\_bj^{**} S T$ 
  using  $assms$  apply (induction)
  apply simp
  by (auto intro:  $rtrancp\_dpll_W-all-inv$   $dpll_W-dpll_W-bj$   $rtrancp.rtrancp\_into-rtrancp$ )

lemma  $rtrancp\_dpll-rtrancp\_dpll_W$ :
  assumes  $dpll_{W-NOT}.dpll\_bj^{**} S T$  and  $dpll_W-all-inv S$ 
  shows  $dpll_W^{**} S T$ 
  using  $assms$  apply (induction)
  apply simp
  by (auto intro:  $dpll_W-bj-dpll$   $rtrancp.rtrancp\_into-rtrancp$   $rtrancp\_dpll_W-all-inv$ )

lemma  $dpll-conclusive-state-correctness$ :
  assumes  $dpll_{W-NOT}.dpll\_bj^{**} ([], N) (M, N)$  and  $conclusive\_dpll_W-state (M, N)$ 
  shows  $M \models_{asm} N \longleftrightarrow satisfiable (set-mset N)$ 
proof –
  have  $dpll_W-all-inv ([], N)$ 
  unfolding  $dpll_W-all-inv-def$  by auto
  show ?thesis
  apply (rule  $dpll_W-conclusive-state-correct$ )
  apply (simp add:  $\langle dpll_W-all-inv ([], N) \rangle assms(1)$   $rtrancp\_dpll-rtrancp\_dpll_W$ )
  using  $assms(2)$  by simp
qed

end
theory  $CDCL-W-Level$ 
imports  $Partial-Annotated-Clausal-Logic$ 
begin

```

17.5.1 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  $get\_rev-level :: ('v, nat, 'a) marked-lits \Rightarrow nat \Rightarrow 'v literal \Rightarrow nat$  where
 $get\_rev-level [] - = 0$  |
 $get\_rev-level (Marked\ l\ level\ \# Ls)\ n\ L =$ 
  (if  $atm-of\ l = atm-of\ L$  then  $level$  else  $get\_rev-level\ Ls\ level\ L$ ) |
 $get\_rev-level (Propagated\ l - \# Ls)\ n\ L =$ 
  (if  $atm-of\ l = atm-of\ L$  then  $n$  else  $get\_rev-level\ Ls\ n\ L$ )

```

abbreviation $get-level\ M\ L \equiv get_rev-level\ (rev\ M)\ 0\ L$

```

lemma  $get\_rev-level-uminus[simp]$ :  $get\_rev-level\ M\ n(-L) = get\_rev-level\ M\ n\ L$ 
by (induct arbitrary:  $n$  rule:  $get\_rev-level.induct$ ) auto

```

```

lemma  $atm-of-notin-get\_rev-level-eq-0[simp]$ :
  assumes  $atm-of\ L \notin atm-of\ ' lits-of-l\ M$ 
  shows  $get\_rev-level\ M\ n\ L = 0$ 
  using  $assms$  by (induct  $M$  arbitrary:  $n$  rule:  $marked-lit-list-induct$ ) auto

```

lemma *get-rev-level-ge-0-atm-of-in*:
assumes *get-rev-level* $M\ n\ L > n$
shows $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$
using *assms* **by** (*induct* M *arbitrary*: n *rule*: *marked-lit-list-induct*) *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 $\text{atm-of } L \notin \text{atm-of ' lits-of-l } M$
shows $\text{get-rev-level } (M @ \text{Marked } K\ i \# M')\ n\ L = \text{get-rev-level } (\text{Marked } K\ i \# M')\ i\ L$
using *assms* **by** (*induct* M *arbitrary*: $n\ i$ *rule*: *marked-lit-list-induct*) *auto*

lemma *get-rev-level-notin-end[simp]*:
assumes $\text{atm-of } L \notin \text{atm-of ' lits-of-l } M'$
shows $\text{get-rev-level } (M @ M')\ n\ L = \text{get-rev-level } M\ n\ L$
using *assms* **by** (*induct* M *arbitrary*: n *rule*: *marked-lit-list-induct*) *auto*

If the literal is at the beginning, then the end can be skipped

lemma *get-rev-level-skip-end[simp]*:
assumes $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$
shows $\text{get-rev-level } (M @ M')\ n\ L = \text{get-rev-level } M\ n\ L$
using *assms* **by** (*induct* *arbitrary*: n *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-beginning*:
assumes $\text{atm-of } L' \neq \text{atm-of (lit-of } K)$
shows $\text{get-level } (K \# M)\ L' = \text{get-level } M\ L'$
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 $\text{get-level } (M @ \text{rev } S)\ L = \text{get-level } M\ L$
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 $\text{get-level } (M @ S)\ L = \text{get-level } M\ L$
using *get-level-skip-beginning-not-marked-rev*[*of* $L\ \text{rev } S\ M$] *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 $\text{get-rev-level } (\text{rev } S @ \text{rev } M)\ 0\ L = \text{get-level } M\ L$
using *get-level-skip-beginning-not-marked-rev*[*of* $L\ \text{rev } S\ M$] *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 $\text{get-rev-level } M\ n\ L = n$
using *assms* **by** (*induction* M *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-all-not-marked[simp]*:

```

fixes M
defines M'  $\equiv$  rev M
assumes  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
shows get-level M L = 0
proof –
  have M: M = rev M'
    unfolding M'-def by auto
  show ?thesis
    using assms unfolding M by (induction M' rule: marked-lit-list-induct) auto
qed

```

abbreviation MMax M \equiv Max (set-mset M)

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

definition get-maximum-level :: ('a, nat, 'b) marked-lit list \Rightarrow 'a literal multiset \Rightarrow nat
where
 get-maximum-level M D = MMax ($\{\#0\# \}$ + image-mset (get-level M) D)

lemma get-maximum-level-ge-get-level:
 $L \in \# D \implies \text{get-maximum-level } M D \geq \text{get-level } M L$
unfolding get-maximum-level-def **by** auto

lemma get-maximum-level-empty[simp]:
 get-maximum-level M $\{\# \}$ = 0
unfolding get-maximum-level-def **by** auto

lemma get-maximum-level-exists-lit-of-max-level:
 $D \neq \{\# \} \implies \exists L \in \# D. \text{get-level } M L = \text{get-maximum-level } M D$
unfolding get-maximum-level-def
apply (induct D)
apply simp
by (rename-tac D x, case-tac D = $\{\# \}$) (auto simp add: max-def)

lemma get-maximum-level-empty-list[simp]:
 get-maximum-level [] D = 0
unfolding get-maximum-level-def **by** (simp add: image-constant-conv)

lemma get-maximum-level-single[simp]:
 get-maximum-level M $\{\# L \# \}$ = get-level M L
unfolding get-maximum-level-def **by** simp

lemma get-maximum-level-plus:
 get-maximum-level M (D + D') = max (get-maximum-level M D) (get-maximum-level M D')
by (induct D) (auto simp add: get-maximum-level-def)

lemma get-maximum-level-exists-lit:
assumes n: $n > 0$
and max: get-maximum-level M D = n
shows $\exists L \in \# D. \text{get-level } M L = n$
proof –
have f: finite (insert 0 (($\lambda L. \text{get-level } M L$) ‘ set-mset D)) **by** auto
then have n \in (($\lambda L. \text{get-level } M L$) ‘ set-mset D)
using n max Max-in[OF f] **unfolding** get-maximum-level-def **by** simp
then show $\exists L \in \# D. \text{get-level } M L = n$ **by** auto

qed

lemma *get-maximum-level-skip-first*[simp]:

assumes *atm-of* $L \notin \text{atms-of } D$

shows *get-maximum-level* (*Propagated* $L \ C \ \# \ M$) $D = \text{get-maximum-level } M \ D$

using *assms* **unfolding** *get-maximum-level-def* *atms-of-def*

atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set

by (*smt* *atm-of-in-atm-of-set-in-uminus* *get-level-skip-beginning* *image-iff* *marked-lit.sel*(2) *multiset.map-cong0*)

lemma *get-maximum-level-skip-beginning*:

assumes $DH: \text{atms-of } D \subseteq \text{atm-of 'lits-of-l } H$

shows *get-maximum-level* ($c \ @ \ \text{Marked } Kh \ i \ \# \ H$) $D = \text{get-maximum-level } H \ D$

proof –

have (*get-rev-level* (*rev* $H \ @ \ \text{Marked } Kh \ i \ \# \ \text{rev } c$) 0) ‘ *set-mset* D

$= (\text{get-rev-level } (\text{rev } H) \ 0) \ \text{‘ set-mset } D$

using DH **unfolding** *atms-of-def*

by (*metis* (*no-types*, *lifting*) *get-rev-level-skip-end* *image-cong* *image-subset-iff* *set-rev*)

then show ?thesis **using** DH **unfolding** *get-maximum-level-def* **by** *auto*

qed

lemma *get-maximum-level-D-single-propagated*:

get-maximum-level [*Propagated* $x21 \ x22$] $D = 0$

proof –

have $A: \text{insert } 0 \ ((\lambda L. \ 0) \ \text{‘ } (\text{set-mset } D \cap \{L. \ \text{atm-of } x21 = \text{atm-of } L\}))$

$\cup (\lambda L. \ 0) \ \text{‘ } (\text{set-mset } D \cap \{L. \ \text{atm-of } x21 \neq \text{atm-of } L\})) = \{0\}$

by *auto*

show ?thesis **unfolding** *get-maximum-level-def* **by** (*simp* *add: A*)

qed

lemma *get-maximum-level-skip-notin*:

assumes $D: \forall L \in \#D. \ \text{atm-of } L \in \text{atm-of 'lits-of-l } M$

shows *get-maximum-level* $M \ D = \text{get-maximum-level } (\text{Propagated } x21 \ x22 \ \# \ M) \ D$

proof –

have $A: (\text{get-rev-level } (\text{rev } M \ @ \ [\text{Propagated } x21 \ x22]) \ 0) \ \text{‘ set-mset } D$

$= (\text{get-rev-level } (\text{rev } M) \ 0) \ \text{‘ set-mset } D$

using D **by** (*auto* *intro!*: *image-cong* *simp* *add: lits-of-def*)

show ?thesis **unfolding** *get-maximum-level-def* **by** (*auto* *simp: A*)

qed

lemma *get-maximum-level-skip-un-marked-not-present*:

assumes $\forall L \in \#D. \ \text{atm-of } L \in \text{atm-of 'lits-of-l } aa$ **and**

$\forall m \in \text{set } M. \ \neg \text{is-marked } m$

shows *get-maximum-level* $aa \ D = \text{get-maximum-level } (M \ @ \ aa) \ D$

using *assms* **by** (*induction* M *rule: marked-lit-list-induct*)

(*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* $K \ i \ \# \ l$) = $\max i \ (\text{get-maximum-possible-level } l) \ |$

get-maximum-possible-level (*Propagated* $- \ - \ \# \ l$) = *get-maximum-possible-level* l

lemma *get-maximum-possible-level-append*[simp]:

get-maximum-possible-level ($M \ @ \ M'$)

$= \max (\text{get-maximum-possible-level } M) \ (\text{get-maximum-possible-level } M')$

by (induct M rule: marked-lit-list-induct) auto

lemma get-maximum-possible-level-rev[simp]:

get-maximum-possible-level (rev M) = get-maximum-possible-level M

by (induct M rule: marked-lit-list-induct) auto

lemma get-maximum-possible-level-ge-get-rev-level:

max (get-maximum-possible-level M) $i \geq$ get-rev-level M i L

by (induct M arbitrary: i rule: marked-lit-list-induct) (auto simp add: le-max-iff-disj)

lemma get-maximum-possible-level-ge-get-level[simp]:

get-maximum-possible-level $M \geq$ get-level M L

using get-maximum-possible-level-ge-get-rev-level[of rev - 0] **by** auto

lemma get-maximum-possible-level-ge-get-maximum-level[simp]:

get-maximum-possible-level $M \geq$ get-maximum-level M D

using get-maximum-level-exists-lit-of-max-level **unfolding** Bex-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

by (induct A rule: marked-lit-list-induct) auto

17.5.2 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 (cases 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:

$i \in \text{set } (\text{get-all-levels-of-marked } M) \longleftrightarrow (\exists c K c'. M = c @ \text{Marked } K i \# c') \text{ (is } ?A \longleftrightarrow ?B)$

proof

assume ?B

then show ?A **by** auto

next

assume ?A

then show ?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:
 $get_rev_level\ M\ n\ L \leq Max\ (set\ (n\ \# \ get_all_levels_of_marked\ M))$
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 $L \in atm-of\ ' \ (lits-of-l\ M)$
shows $get_rev_level\ M\ n\ L \geq Min\ (set\ (n\ \# \ get_all_levels_of_marked\ M))$
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:
 $get_maximum_possible_level\ M = Max\ (insert\ 0\ (set\ (get_all_levels_of_marked\ M)))$
by (induct M rule: marked-lit-list-induct) (auto simp: insert-commute)

lemma get-rev-level-in-levels-of-marked:
 $get_rev_level\ M\ n\ L \in \{0, n\} \cup set\ (get_all_levels_of_marked\ M)$
by (induction M arbitrary: n rule: marked-lit-list-induct) (force simp add: atm-of-eq-atm-of)+

lemma get-rev-level-in-atms-in-levels-of-marked:
 $atm-of\ L \in atm-of\ ' \ (lits-of-l\ M) \implies$
 $get_rev_level\ M\ n\ L \in \{n\} \cup set\ (get_all_levels_of_marked\ M)$
by (induction M arbitrary: n rule: marked-lit-list-induct) (auto simp add: atm-of-eq-atm-of)

lemma get-all-levels-of-marked-no-marked:
 $(\forall l \in set\ Ls. \neg is_marked\ l) \longleftrightarrow get_all_levels_of_marked\ Ls = []$
by (induction Ls) (auto simp add: get-all-levels-of-marked-cons)

lemma get-level-in-levels-of-marked:
 $get_level\ M\ L \in \{0\} \cup set\ (get_all_levels_of_marked\ M)$
using get-rev-level-in-levels-of-marked[of $rev\ M\ 0\ L$] **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 $L \notin atm-of\ ' \ (lits-of-l\ M)$
shows
 $get_level\ (K\ @\ M)\ L = get_rev_level\ (rev\ K)\ (last\ (0\ \# \ get_all_levels_of_marked\ (rev\ M)))\ L$
using assms
proof (induct M arbitrary: K)
case Nil
then show ?case **by** auto
next
case (Cons $a\ M$)
then have $H: \bigwedge K. get_level\ (K\ @\ M)\ L$

```

    = get-rev-level (rev K) (last (0 # get-all-levels-of-marked (rev M))) L
  by auto
have get-level ((K @ [a])@ M) L
  = get-rev-level (a # rev K) (last (0 # get-all-levels-of-marked (rev M))) L
  using H[of K @ [a]] by simp
then show ?case using Cons(2) by (cases a) auto
qed

lemma get-rev-level-can-skip-correctly-ordered:
  assumes
    no-dup M and
    atm-of L  $\notin$  atm-of ' (lits-of-l M) and
    get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))$ ]
  shows get-rev-level (rev M @ K) 0 L = get-rev-level K (length (get-all-levels-of-marked M)) L
  using assms
proof (induct M arbitrary: K rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L' i M K)
  then have
    i: i = Suc (length (get-all-levels-of-marked M)) and
    get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))$ ]
    by auto
  then have get-rev-level (rev M @ (Marked L' i # K)) 0 L
    = get-rev-level (Marked L' i # K) (length (get-all-levels-of-marked M)) L
    using marked by auto
  then show ?case using marked unfolding i by auto
next
  case (proped L' D M K)
  then have get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))$ ]
    by auto
  then have get-rev-level (rev M @ (Propagated L' D # K)) 0 L
    = get-rev-level (Propagated L' D # K) (length (get-all-levels-of-marked M)) L
    using proped by auto
  then show ?case using proped by auto
qed

lemma get-level-skip-beginning-hd-get-all-levels-of-marked:
  assumes atm-of L  $\notin$  atm-of ' lits-of-l S and get-all-levels-of-marked S  $\neq$  []
  shows get-level (M@ S) L = get-rev-level (rev M) (hd (get-all-levels-of-marked S)) L
  using assms
proof (induction S arbitrary: M rule: marked-lit-list-induct)
  case nil
  then show ?case by (auto simp add: lits-of-def)
next
  case (marked K m)
  note notin = this(2)
  then show ?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 *CDCL-Abstract-Clause-Representation List-More CDCL-W-Level Wellfounded-More*

begin

18 Weidenbach's CDCL

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

18.1 The State

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

locale *state_W-ops* =
raw-clss mset-clss union-clss insert-clss remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
 +
raw-clss mset-ccls union-ccls insert-ccls remove-clit
for
 — Clause
mset-clss :: 'cls \Rightarrow 'v clause and
union-clss :: 'cls \Rightarrow 'cls \Rightarrow 'cls and
insert-clss :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and
remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and

 — Multiset of Clauses
mset-clss :: 'clss \Rightarrow 'v clauses and
union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss and
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool and
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and
remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and

mset-ccls :: 'ccls \Rightarrow 'v clause and
union-ccls :: 'ccls \Rightarrow 'ccls \Rightarrow 'ccls and
insert-ccls :: 'v literal \Rightarrow 'ccls \Rightarrow 'ccls and
remove-clit :: 'v literal \Rightarrow 'ccls \Rightarrow 'ccls
 +
fixes
ccls-of-clss :: 'cls \Rightarrow 'ccls and
clss-of-ccls :: 'ccls \Rightarrow 'clss and

trail :: 'st \Rightarrow ('v, nat, 'v clause) marked-lits and
hd-raw-trail :: 'st \Rightarrow ('v, nat, 'cls) marked-lit and
raw-init-clss :: 'st \Rightarrow 'clss and
raw-learned-clss :: 'st \Rightarrow 'clss and
backtrack-lvl :: 'st \Rightarrow nat and
raw-conflicting :: 'st \Rightarrow 'ccls option and

cons-trail :: ('v, nat, 'cls) marked-lit \Rightarrow 'st \Rightarrow 'st and
tl-trail :: 'st \Rightarrow 'st and
add-init-clss :: 'cls \Rightarrow 'st \Rightarrow 'st and
add-learned-clss :: 'cls \Rightarrow 'st \Rightarrow 'st and
remove-clss :: 'cls \Rightarrow 'st \Rightarrow 'st and
update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st and

```

    update-conflicting :: 'cls option  $\Rightarrow$  'st  $\Rightarrow$  'st and

    init-state :: 'clss  $\Rightarrow$  'st and
    restart-state :: 'st  $\Rightarrow$  'st
  assumes
    mset-ccls-ccls-of-cls[simp]:
      mset-ccls (ccls-of-cls C) = mset-cls C and
    mset-cls-cls-of-ccls[simp]:
      mset-cls (cls-of-ccls D) = mset-ccls D and
    ex-mset-cls:  $\exists a. \text{mset-cls } a = E$ 
  begin
  fun mmset-of-mlit :: ('a, 'b, 'cls) marked-lit  $\Rightarrow$  ('a, 'b, 'v clause) marked-lit
    where
    mmset-of-mlit (Propagated L C) = Propagated L (mset-cls C) |
    mmset-of-mlit (Marked L i) = Marked L i

  lemma lit-of-mmset-of-mlit[simp]:
    lit-of (mmset-of-mlit a) = lit-of a
  by (cases a) auto

  lemma lit-of-mmset-of-mlit-set-lit-of-l[simp]:
    lit-of ' mmset-of-mlit ' set M' = lits-of-l M'
  by (induction M') auto

  lemma map-mmset-of-mlit-true-annots-true-cls[simp]:
    map mmset-of-mlit M'  $\models_{as} C \longleftrightarrow M' \models_{as} C$ 
  by (simp add: true-annots-true-cls)

  abbreviation init-clss  $\equiv \lambda S. \text{mset-clss (raw-init-clss } S)$ 
  abbreviation learned-clss  $\equiv \lambda S. \text{mset-clss (raw-learned-clss } S)$ 
  abbreviation conflicting  $\equiv \lambda S. \text{map-option mset-ccls (raw-conflicting } S)$ 

  notation insert-cls (infix !++ 50)

  notation in-clss (infix ! $\in$ ! 50)
  notation union-clss (infix  $\oplus$  50)
  notation insert-clss (infix !++! 50)

  notation union-ccls (infix ! $\cup$  50)

  definition raw-clauses :: 'st  $\Rightarrow$  'clss where
    raw-clauses S = union-clss (raw-init-clss S) (raw-learned-clss S)

  abbreviation clauses :: 'st  $\Rightarrow$  'v clauses where
    clauses S  $\equiv \text{mset-clss (raw-clauses } S)$ 

  end

  locale stateW =
    stateW-ops
    — functions for clauses:
    mset-cls union-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss

    — functions for the conflicting clause:

```

mset-ccls union-ccls insert-ccls remove-clit

— Conversion between conflicting and non-conflicting
ccls-of-cls cls-of-ccls

— functions for the state:

— access functions:

trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

— changing state:

*cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting*

— get state:

init-state

restart-state

for

mset-cls:: 'cls \Rightarrow 'v clause and

union-cls :: 'cls \Rightarrow 'cls \Rightarrow 'cls and

insert-cls :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and

remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and

mset-clss:: 'clss \Rightarrow 'v clauses and

union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss and

in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool and

insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and

remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and

mset-ccls:: 'ccls \Rightarrow 'v clause and

union-ccls :: 'ccls \Rightarrow 'ccls \Rightarrow 'ccls and

insert-ccls :: 'v literal \Rightarrow 'ccls \Rightarrow 'ccls and

remove-clit :: 'v literal \Rightarrow 'ccls \Rightarrow 'ccls and

ccls-of-cls :: 'cls \Rightarrow 'ccls and

cls-of-ccls :: 'ccls \Rightarrow 'cls and

trail :: 'st \Rightarrow ('v, nat, 'v clause) marked-lits and

hd-raw-trail :: 'st \Rightarrow ('v, nat, 'cls) marked-lit and

raw-init-clss :: 'st \Rightarrow 'clss and

raw-learned-clss :: 'st \Rightarrow 'clss and

backtrack-lvl :: 'st \Rightarrow nat and

raw-conflicting :: 'st \Rightarrow 'ccls option and

cons-trail :: ('v, nat, 'cls) marked-lit \Rightarrow 'st \Rightarrow 'st and

tl-trail :: 'st \Rightarrow 'st and

add-init-cls :: 'cls \Rightarrow 'st \Rightarrow 'st and

add-learned-cls :: 'cls \Rightarrow 'st \Rightarrow 'st and

remove-cls :: 'cls \Rightarrow 'st \Rightarrow 'st and

update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st and

update-conflicting :: 'ccls option \Rightarrow 'st \Rightarrow 'st and

init-state :: 'clss \Rightarrow 'st and

restart-state :: 'st \Rightarrow 'st +

assumes

hd-raw-trail: trail S \neq [] \implies mmset-of-mlit (hd-raw-trail S) = hd (trail S) and

trail-cons-trail[simp]:

$\bigwedge L \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } L) \implies$
 $trail (cons\text{-}trail \text{ L st}) = mset\text{-of}\text{-}mlit \text{ L } \# trail \text{ st and}$
 $trail\text{-}tl\text{-}trail[simp]: \bigwedge st. trail (tl\text{-}trail \text{ st}) = tl (trail \text{ st}) \text{ and}$
 $trail\text{-}add\text{-}init\text{-}cls[simp]:$
 $\bigwedge st \text{ C. no-dup } (trail \text{ st}) \implies trail (add\text{-}init\text{-}cls \text{ C st}) = trail \text{ st and}$
 $trail\text{-}add\text{-}learned\text{-}cls[simp]:$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies trail (add\text{-}learned\text{-}cls \text{ C st}) = trail \text{ st and}$
 $trail\text{-}remove\text{-}cls[simp]:$
 $\bigwedge C \text{ st. trail } (remove\text{-}cls \text{ C st}) = trail \text{ st and}$
 $trail\text{-}update\text{-}backtrack\text{-}lvl[simp]: \bigwedge st \text{ C. trail } (update\text{-}backtrack\text{-}lvl \text{ C st}) = trail \text{ st and}$
 $trail\text{-}update\text{-}conflicting[simp]: \bigwedge C \text{ st. trail } (update\text{-}conflicting \text{ C st}) = trail \text{ st and}$

$init\text{-}clss\text{-}cons\text{-}trail[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $raw\text{-}init\text{-}clss (cons\text{-}trail \text{ M st}) = raw\text{-}init\text{-}clss \text{ st}$
and
 $init\text{-}clss\text{-}tl\text{-}trail[simp]:$
 $\bigwedge st. raw\text{-}init\text{-}clss (tl\text{-}trail \text{ st}) = raw\text{-}init\text{-}clss \text{ st and}$
 $init\text{-}clss\text{-}add\text{-}init\text{-}cls[simp]:$
 $\bigwedge st \text{ C. no-dup } (trail \text{ st}) \implies init\text{-}clss (add\text{-}init\text{-}cls \text{ C st}) = \{\#mset\text{-}cls \text{ C}\# \} + init\text{-}clss \text{ st}$
and
 $init\text{-}clss\text{-}add\text{-}learned\text{-}cls[simp]:$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies raw\text{-}init\text{-}clss (add\text{-}learned\text{-}cls \text{ C st}) = raw\text{-}init\text{-}clss \text{ st and}$
 $init\text{-}clss\text{-}remove\text{-}cls[simp]:$
 $\bigwedge C \text{ st. init\text{-}clss } (remove\text{-}cls \text{ C st}) = removeAll\text{-}mset (mset\text{-}cls \text{ C}) (init\text{-}clss \text{ st}) \text{ and}$
 $init\text{-}clss\text{-}update\text{-}backtrack\text{-}lvl[simp]:$
 $\bigwedge st \text{ C. raw\text{-}init\text{-}clss } (update\text{-}backtrack\text{-}lvl \text{ C st}) = raw\text{-}init\text{-}clss \text{ st and}$
 $init\text{-}clss\text{-}update\text{-}conflicting[simp]:$
 $\bigwedge C \text{ st. raw\text{-}init\text{-}clss } (update\text{-}conflicting \text{ C st}) = raw\text{-}init\text{-}clss \text{ st and}$

$learned\text{-}clss\text{-}cons\text{-}trail[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $raw\text{-}learned\text{-}clss (cons\text{-}trail \text{ M st}) = raw\text{-}learned\text{-}clss \text{ st and}$
 $learned\text{-}clss\text{-}tl\text{-}trail[simp]:$
 $\bigwedge st. raw\text{-}learned\text{-}clss (tl\text{-}trail \text{ st}) = raw\text{-}learned\text{-}clss \text{ st and}$
 $learned\text{-}clss\text{-}add\text{-}init\text{-}cls[simp]:$
 $\bigwedge st \text{ C. no-dup } (trail \text{ st}) \implies raw\text{-}learned\text{-}clss (add\text{-}init\text{-}cls \text{ C st}) = raw\text{-}learned\text{-}clss \text{ st and}$
 $learned\text{-}clss\text{-}add\text{-}learned\text{-}cls[simp]:$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies$
 $learned\text{-}clss (add\text{-}learned\text{-}cls \text{ C st}) = \{\#mset\text{-}cls \text{ C}\# \} + learned\text{-}clss \text{ st and}$
 $learned\text{-}clss\text{-}remove\text{-}cls[simp]:$
 $\bigwedge C \text{ st. learned\text{-}clss } (remove\text{-}cls \text{ C st}) = removeAll\text{-}mset (mset\text{-}cls \text{ C}) (learned\text{-}clss \text{ st}) \text{ and}$
 $learned\text{-}clss\text{-}update\text{-}backtrack\text{-}lvl[simp]:$
 $\bigwedge st \text{ C. raw\text{-}learned\text{-}clss } (update\text{-}backtrack\text{-}lvl \text{ C st}) = raw\text{-}learned\text{-}clss \text{ st and}$
 $learned\text{-}clss\text{-}update\text{-}conflicting[simp]:$
 $\bigwedge C \text{ st. raw\text{-}learned\text{-}clss } (update\text{-}conflicting \text{ C st}) = raw\text{-}learned\text{-}clss \text{ st and}$

$backtrack\text{-}lvl\text{-}cons\text{-}trail[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $backtrack\text{-}lvl (cons\text{-}trail \text{ M st}) = backtrack\text{-}lvl \text{ st and}$
 $backtrack\text{-}lvl\text{-}tl\text{-}trail[simp]:$
 $\bigwedge st. backtrack\text{-}lvl (tl\text{-}trail \text{ st}) = backtrack\text{-}lvl \text{ st and}$
 $backtrack\text{-}lvl\text{-}add\text{-}init\text{-}cls[simp]:$
 $\bigwedge st \text{ C. no-dup } (trail \text{ st}) \implies backtrack\text{-}lvl (add\text{-}init\text{-}cls \text{ C st}) = backtrack\text{-}lvl \text{ st and}$
 $backtrack\text{-}lvl\text{-}add\text{-}learned\text{-}cls[simp]:$

$\bigwedge C \text{ st. no-dup } (\text{trail } st) \implies \text{backtrack-lvl } (\text{add-learned-cls } C \text{ st}) = \text{backtrack-lvl } st$ **and**
 $\text{backtrack-lvl-remove-cls}[simp]:$
 $\bigwedge C \text{ st. backtrack-lvl } (\text{remove-cls } C \text{ st}) = \text{backtrack-lvl } st$ **and**
 $\text{backtrack-lvl-update-backtrack-lvl}[simp]:$
 $\bigwedge st \ k. \text{backtrack-lvl } (\text{update-backtrack-lvl } k \text{ st}) = k$ **and**
 $\text{backtrack-lvl-update-conflicting}[simp]:$
 $\bigwedge C \text{ st. backtrack-lvl } (\text{update-conflicting } C \text{ st}) = \text{backtrack-lvl } st$ **and**

 $\text{conflicting-cons-trail}[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (\text{trail } st) \ (\text{lit-of } M) \implies$
 $\text{raw-conflicting } (\text{cons-trail } M \text{ st}) = \text{raw-conflicting } st$ **and**
 $\text{conflicting-tl-trail}[simp]:$
 $\bigwedge st. \text{raw-conflicting } (\text{tl-trail } st) = \text{raw-conflicting } st$ **and**
 $\text{conflicting-add-init-cls}[simp]:$
 $\bigwedge st \ C. \text{no-dup } (\text{trail } st) \implies \text{raw-conflicting } (\text{add-init-cls } C \text{ st}) = \text{raw-conflicting } st$ **and**
 $\text{conflicting-add-learned-cls}[simp]:$
 $\bigwedge C \text{ st. no-dup } (\text{trail } st) \implies \text{raw-conflicting } (\text{add-learned-cls } C \text{ st}) = \text{raw-conflicting } st$
and
 $\text{conflicting-remove-cls}[simp]:$
 $\bigwedge C \text{ st. raw-conflicting } (\text{remove-cls } C \text{ st}) = \text{raw-conflicting } st$ **and**
 $\text{conflicting-update-backtrack-lvl}[simp]:$
 $\bigwedge st \ C. \text{raw-conflicting } (\text{update-backtrack-lvl } C \text{ st}) = \text{raw-conflicting } st$ **and**
 $\text{conflicting-update-conflicting}[simp]:$
 $\bigwedge C \text{ st. raw-conflicting } (\text{update-conflicting } C \text{ st}) = C$ **and**

 $\text{init-state-trail}[simp]: \bigwedge N. \text{trail } (\text{init-state } N) = []$ **and**
 $\text{init-state-clss}[simp]: \bigwedge N. (\text{init-clss } (\text{init-state } N)) = \text{mset-clss } N$ **and**
 $\text{init-state-learned-clss}[simp]: \bigwedge N. \text{learned-clss } (\text{init-state } N) = \{\#\}$ **and**
 $\text{init-state-backtrack-lvl}[simp]: \bigwedge N. \text{backtrack-lvl } (\text{init-state } N) = 0$ **and**
 $\text{init-state-conflicting}[simp]: \bigwedge N. \text{conflicting } (\text{init-state } N) = \text{None}$ **and**

 $\text{trail-restart-state}[simp]: \text{trail } (\text{restart-state } S) = []$ **and**
 $\text{init-clss-restart-state}[simp]: \text{init-clss } (\text{restart-state } S) = \text{init-clss } S$ **and**
 $\text{learned-clss-restart-state}[intro]:$
 $\text{learned-clss } (\text{restart-state } S) \subseteq \# \text{ learned-clss } S$ **and**
 $\text{backtrack-lvl-restart-state}[simp]: \text{backtrack-lvl } (\text{restart-state } S) = 0$ **and**
 $\text{conflicting-restart-state}[simp]: \text{conflicting } (\text{restart-state } S) = \text{None}$

begin

lemma

shows

$\text{clauses-cons-trail}[simp]:$
 $\text{undefined-lit } (\text{trail } S) \ (\text{lit-of } M) \implies \text{clauses } (\text{cons-trail } M \ S) = \text{clauses } S$ **and**

 $\text{clss-tl-trail}[simp]: \text{clauses } (\text{tl-trail } S) = \text{clauses } S$ **and**
 $\text{clauses-add-learned-clss-unfolded}: \text{no-dup } (\text{trail } S) \implies \text{clauses } (\text{add-learned-clss } U \ S) =$
 $\{\#\text{mset-clss } U \#\} + \text{learned-clss } S + \text{init-clss } S$
and
 $\text{clauses-add-init-clss}[simp]:$
 $\text{no-dup } (\text{trail } S) \implies$
 $\text{clauses } (\text{add-init-clss } N \ S) = \{\#\text{mset-clss } N \#\} + \text{init-clss } S + \text{learned-clss } S$ **and**
 $\text{clauses-update-backtrack-lvl}[simp]: \text{clauses } (\text{update-backtrack-lvl } k \ S) = \text{clauses } S$ **and**
 $\text{clauses-update-conflicting}[simp]: \text{clauses } (\text{update-conflicting } D \ S) = \text{clauses } S$ **and**

clauses-remove-cls[simp]:
clauses (*remove-cls* *C S*) = *removeAll-mset* (*mset-cls* *C*) (*clauses* *S*) **and**
clauses-add-learned-cls[simp]:
no-dup (*trail* *S*) \implies *clauses* (*add-learned-cls* *C S*) = {#*mset-cls* *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*) = *mset-clss* *N*
prefer 9 using *raw-clauses-def* *learned-clss-restart-state* **apply** *fastforce*
by (*auto simp: ac-simps replicate-mset-plus raw-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 option **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* \implies *T* \sim *U* \implies *S* \sim *U*
unfolding *state-eq-def* **by** *auto*

lemma
shows
state-eq-trail: *S* \sim *T* \implies *trail* *S* = *trail* *T* **and**
state-eq-init-clss: *S* \sim *T* \implies *init-clss* *S* = *init-clss* *T* **and**
state-eq-learned-clss: *S* \sim *T* \implies *learned-clss* *S* = *learned-clss* *T* **and**
state-eq-backtrack-lvl: *S* \sim *T* \implies *backtrack-lvl* *S* = *backtrack-lvl* *T* **and**
state-eq-conflicting: *S* \sim *T* \implies *conflicting* *S* = *conflicting* *T* **and**
state-eq-clauses: *S* \sim *T* \implies *clauses* *S* = *clauses* *T* **and**
state-eq-undefined-lit: *S* \sim *T* \implies *undefined-lit* (*trail* *S*) *L* = *undefined-lit* (*trail* *T*) *L*
unfolding *state-eq-def* *raw-clauses-def* **by** *auto*

lemma *state-eq-raw-conflicting-None*:
S \sim *T* \implies *conflicting* *T* = None \implies *raw-conflicting* *S* = None
unfolding *state-eq-def* *raw-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*
state-eq-raw-conflicting-None

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*[*intro*]:
 $x \in \text{atms-of-mm } (\text{learned-clss } (\text{restart-state } S)) \implies x \in \text{atms-of-mm } (\text{learned-clss } S)$
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* $F (\text{tl-trail } S)$)
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 $[]::('v, \text{nat}, 'v \text{ clause})$ 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 $\text{clauses } (\text{reduce-trail-to } ([]::'a \text{ list}) Sa) = \text{clauses } Sa$
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 $\text{trail } S = F' @ F$
shows $\text{trail } (\text{reduce-trail-to } F S) = F$
using *assms* **by** (induction F' arbitrary: S) (auto *simp*: *reduce-trail-to-length-ne*)

lemma *clauses-reduce-trail-to*[*simp*]:
 $\text{clauses } (\text{reduce-trail-to } F S) = \text{clauses } S$
apply (induction $F S$ rule: *reduce-trail-to.induct*)
by (*metis* *clss-tl-trail reduce-trail-to.simps*)

lemma *conflicting-update-trial*[*simp*]:
 $\text{conflicting } (\text{reduce-trail-to } F S) = \text{conflicting } S$

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 # []]*)
by (*cases F'*) (*auto simp add:tl-append reduce-trail-to-skip-beginning*)

lemma *reduce-trail-to-add-learned-clss[simp]*:
no-dup (trail S) \implies
trail (reduce-trail-to F (add-learned-clss C S)) = trail (reduce-trail-to F S)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-add-init-clss[simp]*:
no-dup (trail S) \implies
trail (reduce-trail-to F (add-init-clss C S)) = trail (reduce-trail-to F S)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-remove-learned-clss[simp]*:
trail (reduce-trail-to F (remove-clss 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) ∈ set (get-all-marked-decomposition M)
  shows a = [] ∨ (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 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)
  by (simp add: reduce-trail-to.simps)

lemma trail-reduce-trail-to-drop:
  trail (reduce-trail-to F S) =
    (if length (trail S) ≥ length F
     then drop (length (trail S) - length F) (trail S)
     else [])
  apply (induction F S rule: reduce-trail-to.induct)
  apply (rename-tac F S, case-tac trail S)
  apply auto[]
  apply (rename-tac list, case-tac Suc (length list) > length F)
  prefer 2 apply (metis diff-is-0-eq drop-Cons' length-Cons nat-le-linear nat-less-le
    reduce-trail-to-eq-length trail-reduce-trail-to-length-le)
  apply (subgoal-tac Suc (length list) - length F = Suc (length list - length F))
  by (auto simp add: reduce-trail-to-length-ne)

lemma in-get-all-marked-decomposition-trail-update-trail[simp]:
  assumes H: (L # M1, M2) ∈ 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

```

end — end of *state_W* locale

18.2 CDCL Rules

Because of the strategy we will later use, we distinguish propagate, conflict from the other rules

```
locale conflict-driven-clause-learningW =
  stateW
  — functions for clauses:
  mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss

  — functions for the conflicting clause:
  mset-ccls union-ccls insert-ccls remove-clit

  — conversion
  ccls-of-cls cls-of-ccls

  — functions for the state:
  — access functions:
  trail hd-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
  — changing state:
  cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — get state:
  init-state
  restart-state
for
  mset-cls :: 'cls ⇒ 'v clause and
  union-cls :: 'cls ⇒ 'cls ⇒ 'cls and
  insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls and
  remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls and

  mset-clss :: 'clss ⇒ 'v clauses and
  union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
  in-clss :: 'cls ⇒ 'clss ⇒ bool and
  insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and

  mset-ccls :: 'ccls ⇒ 'v clause and
  union-ccls :: 'ccls ⇒ 'ccls ⇒ 'ccls and
  insert-ccls :: 'v literal ⇒ 'ccls ⇒ 'ccls and
  remove-clit :: 'v literal ⇒ 'ccls ⇒ 'ccls and

  ccls-of-cls :: 'cls ⇒ 'ccls and
  cls-of-ccls :: 'ccls ⇒ 'cls and

  trail :: 'st ⇒ ('v, nat, 'v clause) marked-lits and
  hd-trail :: 'st ⇒ ('v, nat, 'cls) marked-lit and
  raw-init-clss :: 'st ⇒ 'clss and
  raw-learned-clss :: 'st ⇒ 'clss and
  backtrack-lvl :: 'st ⇒ nat and
  raw-conflicting :: 'st ⇒ 'ccls option and
```

cons-trail :: ('v, nat, 'cls) marked-lit ⇒ 'st ⇒ 'st **and**
tl-trail :: 'st ⇒ 'st **and**
add-init-cls :: 'cls ⇒ 'st ⇒ 'st **and**
add-learned-cls :: 'cls ⇒ 'st ⇒ 'st **and**
remove-cls :: 'cls ⇒ 'st ⇒ 'st **and**
update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st **and**
update-conflicting :: 'ccls option ⇒ 'st ⇒ 'st **and**

init-state :: 'cls ⇒ 'st **and**
restart-state :: 'st ⇒ 'st

begin

inductive *propagate* :: 'st ⇒ 'st ⇒ bool **for** *S* :: 'st **where**

propagate-rule: *conflicting S = None* ⇒

E !∈! *raw-clauses S* ⇒

L ∈# *mset-cls E* ⇒

trail S ⊨_{as} *CNot (mset-cls (remove-lit L E))* ⇒

undefined-lit (trail S) L ⇒

T ~ *cons-trail (Propagated L E) S* ⇒

propagate S T

inductive-cases *propagateE*: *propagate S T*

inductive *conflict* :: 'st ⇒ 'st ⇒ bool **for** *S* :: 'st **where**

conflict-rule:

conflicting S = None ⇒

D !∈! *raw-clauses S* ⇒

trail S ⊨_{as} *CNot (mset-cls D)* ⇒

T ~ *update-conflicting (Some (ccls-of-cls D)) S* ⇒

conflict S T

inductive-cases *conflictE*: *conflict S T*

inductive *backtrack* :: 'st ⇒ 'st ⇒ bool **for** *S* :: 'st **where**

backtrack-rule:

raw-conflicting S = Some D ⇒

L ∈# *mset-ccls D* ⇒

(*Marked K (i+1) # M1, M2*) ∈ *set (get-all-marked-decomposition (trail S))* ⇒

get-level (trail S) L = backtrack-lvl S ⇒

get-level (trail S) L = get-maximum-level (trail S) (mset-ccls D) ⇒

get-maximum-level (trail S) (mset-ccls (remove-clit L D)) ≡ *i* ⇒

T ~ *cons-trail (Propagated L (cls-of-ccls D))*

(*reduce-trail-to M1*

(*add-learned-cls (cls-of-ccls D)*

(*update-backtrack-lvl i*

(*update-conflicting None S*)))) ⇒

backtrack S T

inductive-cases *backtrackE*: *backtrack S T*

thm *backtrackE*

inductive *decide* :: 'st ⇒ 'st ⇒ bool **for** *S* :: 'st **where**

decide-rule:

conflicting S = None ⇒

undefined-lit (trail S) L ⇒

$atm-of\ L \in atm\text{-}of\text{-}mm\ (init\text{-}cls\ S) \implies$
 $T \sim cons\text{-}trail\ (Marked\ L\ (backtrack\text{-}lvl\ S + 1))\ (incr\text{-}lvl\ S) \implies$
 $decide\ S\ T$

inductive-cases *decideE*: *decide S T*

inductive *skip* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

skip-rule:

$trail\ S = Propagated\ L\ C' \# M \implies$
 $raw\text{-}conflicting\ S = Some\ E \implies$
 $-L \notin \# mset\text{-}ccls\ E \implies$
 $mset\text{-}ccls\ E \neq \{\#\} \implies$
 $T \sim tl\text{-}trail\ S \implies$
 $skip\ S\ T$

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

get-maximum-level (*Propagated L (C + {#L#}) # M*) *D* = *k* \vee *k* = 0 (that was in a previous version of the book) is equivalent to *get-maximum-level* (*Propagated L (C + {#L#}) # M*) *D* = *k*, when the structural invariants holds.

inductive *resolve* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

resolve-rule: $trail\ S \neq [] \implies$

$hd\text{-}raw\text{-}trail\ S = Propagated\ L\ E \implies$
 $L \in \# mset\text{-}cls\ E \implies$
 $raw\text{-}conflicting\ S = Some\ D' \implies$
 $-L \in \# mset\text{-}ccls\ D' \implies$
 $get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ (remove\text{-}clit\ (-L)\ D')) = backtrack\text{-}lvl\ S \implies$
 $T \sim update\text{-}conflicting\ (Some\ (union\text{-}ccls\ (remove\text{-}clit\ (-L)\ D')\ (ccls\text{-}of\text{-}cls\ (remove\text{-}lit\ L\ E))))$
 $(tl\text{-}trail\ S) \implies$
 $resolve\ S\ T$

inductive-cases *resolveE*: *resolve S T*

inductive *restart* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

$restart: state\ S = (M, N, U, k, None) \implies \neg M \models_{asm} clauses\ S$
 $\implies T \sim restart\text{-}state\ S$
 $\implies restart\ S\ T$

inductive-cases *restartE*: *restart S T*

We add the condition $C \notin \# init\text{-}cls\ S$, to maintain consistency even without the strategy.

inductive *forget*:: 'st \Rightarrow 'st \Rightarrow bool **where**

forget-rule:

$conflicting\ S = None \implies$
 $C !\in ! raw\text{-}learned\text{-}cls\ S \implies$
 $\neg(trail\ S) \models_{asm} clauses\ S \implies$
 $mset\text{-}cls\ C \notin set\ (get\text{-}all\text{-}mark\text{-}of\text{-}propagated\ (trail\ S)) \implies$
 $mset\text{-}cls\ C \notin \# init\text{-}cls\ S \implies$
 $T \sim remove\text{-}cls\ C\ S \implies$
 $forget\ S\ T$

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

inductive *cdcl_W-rf* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

restart: $\text{restart } S \ T \implies \text{cdcl}_W\text{-rf } S \ T \mid$
forget: $\text{forget } S \ T \implies \text{cdcl}_W\text{-rf } S \ T$

inductive $\text{cdcl}_W\text{-bj} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**
skip: $\text{skip } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S' \mid$
resolve: $\text{resolve } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S' \mid$
backtrack: $\text{backtrack } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S'$

inductive-cases $\text{cdcl}_W\text{-bjE}$: $\text{cdcl}_W\text{-bj } S \ T$

inductive $\text{cdcl}_W\text{-o} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
decide: $\text{decide } S \ S' \implies \text{cdcl}_W\text{-o } S \ S' \mid$
bj: $\text{cdcl}_W\text{-bj } S \ S' \implies \text{cdcl}_W\text{-o } S \ S'$

inductive $\text{cdcl}_W :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
propagate: $\text{propagate } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$
conflict: $\text{conflict } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$
other: $\text{cdcl}_W\text{-o } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$
rf: $\text{cdcl}_W\text{-rf } S \ S' \implies \text{cdcl}_W \ S \ S'$

lemma *rtranclp-propagate-is-rtranclp-cdcl_W*:
 $\text{propagate}^{**} \ S \ S' \implies \text{cdcl}_W^{**} \ S \ S'$
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
apply (*frule* *propagate*)
using *rtranclp-trans*[*of cdcl_W*] **by** *blast*

lemma *cdcl_W-all-rules-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

fixes $S :: 'st$
assumes
 $\text{cdcl}_W: \text{cdcl}_W \ S \ S'$ **and**
propagate: $\bigwedge T. \text{propagate } S \ T \implies P \ S \ T$ **and**
conflict: $\bigwedge T. \text{conflict } S \ T \implies P \ S \ T$ **and**
forget: $\bigwedge T. \text{forget } S \ T \implies P \ S \ T$ **and**
restart: $\bigwedge T. \text{restart } S \ T \implies P \ S \ T$ **and**
decide: $\bigwedge T. \text{decide } S \ T \implies P \ S \ T$ **and**
skip: $\bigwedge T. \text{skip } S \ T \implies P \ S \ T$ **and**
resolve: $\bigwedge T. \text{resolve } S \ T \implies P \ S \ T$ **and**
backtrack: $\bigwedge T. \text{backtrack } S \ T \implies P \ S \ T$
shows $P \ S \ S'$
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: cdclW-bj.induct) auto
qed
next
case (rf S')
then show ?case
  by (induct rule: cdclW-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

$cdcl_W: cdcl_W\ S\ S' \text{ and}$

$propagateH: \bigwedge C\ L\ T. \text{ conflicting } S = \text{None} \implies$

$C \in \# \text{ raw-clauses } S \implies$

$L \in \# \text{ mset-cls } C \implies$

$\text{trail } S \models_{as} C \text{Not } (\text{remove1-mset } L\ (\text{mset-cls } C)) \implies$

$\text{undefined-lit } (\text{trail } S)\ L \implies$

$T \sim \text{cons-trail } (\text{Propagated } L\ C)\ S \implies$

$P\ S\ T \text{ and}$

$\text{conflictH: } \bigwedge D\ T. \text{ conflicting } S = \text{None} \implies$

$D \in \# \text{ raw-clauses } S \implies$

$\text{trail } S \models_{as} C \text{Not } (\text{mset-cls } D) \implies$

$T \sim \text{update-conflicting } (\text{Some } (\text{ccls-of-cls } D))\ S \implies$

$P\ S\ T \text{ and}$

$\text{forgetH: } \bigwedge C\ U\ T. \text{ conflicting } S = \text{None} \implies$

$C \in \# \text{ raw-learned-clss } S \implies$

$\neg(\text{trail } S) \models_{asm} \text{clauses } S \implies$

$\text{mset-cls } C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \implies$

$\text{mset-cls } C \notin \# \text{ init-clss } S \implies$

$T \sim \text{remove-cls } C\ S \implies$

$P\ S\ T \text{ and}$

$\text{restartH: } \bigwedge T. \neg \text{trail } S \models_{asm} \text{clauses } S \implies$

$\text{conflicting } S = \text{None} \implies$

$T \sim \text{restart-state } S \implies$

$P\ S\ T \text{ and}$

$\text{decideH: } \bigwedge L\ T. \text{ conflicting } S = \text{None} \implies$

$\text{undefined-lit } (\text{trail } S)\ L \implies$

$\text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S) \implies$

$T \sim \text{cons-trail } (\text{Marked } L\ (\text{backtrack-lvl } S + 1))\ (\text{incr-lvl } S) \implies$

$P\ S\ T \text{ and}$

$\text{skipH: } \bigwedge L\ C'\ M\ E\ T.$

$\text{trail } S = \text{Propagated } L\ C' \# M \implies$

$\text{raw-conflicting } S = \text{Some } E \implies$

$\neg L \notin \# \text{ mset-ccls } E \implies \text{mset-ccls } E \neq \{\#\} \implies$

$T \sim \text{tl-trail } S \implies$

$P\ S\ T \text{ and}$

$\text{resolveH: } \bigwedge L\ E\ M\ D\ T.$

$\text{trail } S = \text{Propagated } L\ (\text{mset-cls } E) \# M \implies$

$L \in \# \text{ mset-cls } E \implies$

$\text{hd-raw-trail } S = \text{Propagated } L\ E \implies$

$\text{raw-conflicting } S = \text{Some } D \implies$

$\neg L \in \# \text{ mset-ccls } D \implies$

$\text{get-maximum-level } (\text{trail } S)\ (\text{mset-ccls } (\text{remove-clit } (\neg L)\ D)) = \text{backtrack-lvl } S \implies$

```

  T ~ update-conflicting (Some (union-ccls (remove-clit (-L) D) (ccls-of-cls (remove-lit L E))))
    (tl-trail S) ==>
  P S T and
  backtrackH:  $\bigwedge L D K i M1 M2 T.$ 
  raw-conflicting S = Some D ==>
  L  $\in \#$  mset-ccls D ==>
  (Marked K (i+1) # M1, M2)  $\in$  set (get-all-marked-decomposition (trail S)) ==>
  get-level (trail S) L = backtrack-lvl S ==>
  get-level (trail S) L = get-maximum-level (trail S) (mset-ccls D) ==>
  get-maximum-level (trail S) (remove1-mset L (mset-ccls D))  $\equiv$  i ==>
  T ~ cons-trail (Propagated L (cls-of-ccls D))
    (reduce-trail-to M1
      (add-learned-cls (cls-of-ccls D)
        (update-backtrack-lvl i
          (update-conflicting None S)))) ==>

  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 (auto elim!: propagateE intro!: propagateH)
next
  case (conflict S')
  then show ?case
    by (auto elim!: conflictE intro!: conflictH)
next
  case (restart S')
  then show ?case
    by (auto elim!: restartE intro!: restartH)
next
  case (decide T)
  then show ?case
    by (auto elim!: decideE intro!: decideH)
next
  case (backtrack S')
  then show ?case by (auto elim!: backtrackE intro!: backtrackH
    simp del: state-simp simp add: state-eq-def)
next
  case (forget S')
  then show ?case by (auto elim!: forgetE intro!: forgetH)
next
  case (skip S')
  then show ?case by (auto elim!: skipE intro!: skipH)
next
  case (resolve S')
  then show ?case
    using hd-raw-trail[of S] by (cases trail S) (auto elim!: resolveE intro!: resolveH)
qed

lemma cdclW-o-induct[consumes 1, case-names decide skip resolve backtrack]:
  fixes S :: 'st
  assumes cdclW: cdclW-o S T and
  decideH:  $\bigwedge L T.$  conflicting S = None ==> undefined-lit (trail S) L
    ==> atm-of L  $\in$  atms-of-mm (init-cls S)

```

$\implies T \sim \text{cons-trail } (\text{Marked } L \ (\text{backtrack-lvl } S + 1)) \ (\text{incr-lvl } S)$
 $\implies P \ S \ T \text{ and}$
skipH: $\bigwedge L \ C' \ M \ E \ T.$
 $\text{trail } S = \text{Propagated } L \ C' \ \# \ M \implies$
 $\text{raw-conflicting } S = \text{Some } E \implies$
 $-L \notin \# \ \text{mset-ccls } E \implies \text{mset-ccls } E \neq \{\#\} \implies$
 $T \sim \text{tl-trail } S \implies$
 $P \ S \ T \text{ and}$
resolveH: $\bigwedge L \ E \ M \ D \ T.$
 $\text{trail } S = \text{Propagated } L \ (\text{mset-clc } E) \ \# \ M \implies$
 $L \in \# \ \text{mset-clc } E \implies$
 $\text{hd-raw-trail } S = \text{Propagated } L \ E \implies$
 $\text{raw-conflicting } S = \text{Some } D \implies$
 $-L \in \# \ \text{mset-ccls } D \implies$
 $\text{get-maximum-level } (\text{trail } S) \ (\text{mset-ccls } (\text{remove-clit } (-L) \ D)) = \text{backtrack-lvl } S \implies$
 $T \sim \text{update-conflicting } (\text{Some } (\text{union-ccls } (\text{remove-clit } (-L) \ D) \ (\text{ccls-of-clc } (\text{remove-lit } L \ E))))$
 $(\text{tl-trail } S) \implies$
 $P \ S \ T \text{ and}$
backtrackH: $\bigwedge L \ D \ K \ i \ M1 \ M2 \ T.$
 $\text{raw-conflicting } S = \text{Some } D \implies$
 $L \in \# \ \text{mset-ccls } D \implies$
 $(\text{Marked } K \ (i+1) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \implies$
 $\text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S \implies$
 $\text{get-level } (\text{trail } S) \ L = \text{get-maximum-level } (\text{trail } S) \ (\text{mset-ccls } D) \implies$
 $\text{get-maximum-level } (\text{trail } S) \ (\text{remove1-mset } L \ (\text{mset-ccls } D)) \equiv i \implies$
 $T \sim \text{cons-trail } (\text{Propagated } L \ (\text{cls-of-ccls } D))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-clc } (\text{cls-of-ccls } D)$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting } \text{None } S)))) \implies$
 $P \ S \ T$
shows $P \ S \ T$
using *cdcl_W* **apply** (*induct* T *rule*: *cdcl_W-o.induct*)
using *assms*(2) **apply** (*auto elim*: *decideE*)[1]
apply (*elim* *cdcl_W-bjE* *skipE* *resolveE* *backtrackE*)
apply (*frule* *skipH*; *simp*)
using *hd-raw-trail*[of S] **apply** (*cases* $\text{trail } S$; *auto elim*!: *resolveE* *intro*!: *resolveH*)
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**

$\bigwedge T. \text{decide } S \ T \implies P \ S \ T$ **and**

$\bigwedge T. \text{backtrack } S \ T \implies P \ S \ T$ **and**

$\bigwedge T. \text{skip } S \ T \implies P \ S \ T$ **and**

$\bigwedge T. \text{resolve } S \ T \implies P \ S \ T$

shows $P \ S \ T$

using *assms* **by** (*induct* T *rule*: *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)

```

18.3 Invariants

18.3.1 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 (trail S) L = 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-l M1
proof (rule ccontr)
  let ?M = trail S
  assume L-in-M1:  $\neg \text{atm-of } L \notin \text{atm-of ' lits-of-l } M1$ 
  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-l c
  using L-in-M1 no-dup unfolding Mc lits-of-def by force
  have g-M-eq-g-M1: get-level ?M L = get-level M1 L
  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 simp: rev-swap[symmetric]
    dest: append-cons-eq-upt-length-i)
  then have Max (set (0 # get-all-levels-of-marked (rev M1))) < Suc i by auto
  then have get-level M1 L < Suc i
  using get-rev-level-less-max-get-all-levels-of-marked[of rev M1 0 L] by linarith
  moreover have Suc i  $\leq$  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: cdclW-all-induct)
case (backtrack L D K i M1 M2 T) note decomp = this(3) and L = this(4) and T = this(7) and
  n-d = this(8)
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  $\notin (\lambda l. \text{atm-of } (\text{lit-of } l))$  ‘ set M1
  using backtrack-lit-skipped[of S L K i M1 M2] L decomp backtrack.prems
  by (fastforce simp: 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: defined-lit-map)

```

lemma *cdcl_W-consistent-inv-2*:

```

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+length (get-all-levels-of-marked (trail S))]
shows consistent-interp (lits-of-l (trail S'))
using cdclW-distinctinv-1[OF assms] distinct-consistent-interp by fast

```

lemma *cdcl_W-o-bt*:

```

assumes
  cdclW-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 L D K i M1 M2 T) note decomp = this(3) and T = this(7) and level = this(9)
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  $\notin (\lambda l. \text{atm-of } (\text{lit-of } l))$  ‘ set M1
  using backtrack-lit-skipped[of S L K i M1 M2] backtrack(4,8,9) 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
  cdclW-rf S S' and
  backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and
  get-all-levels-of-marked (trail S) = rev [1..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: cdclW-rf.induct) (auto elim: restartE forgetE)

```

```

lemma cdclW-bt:
  assumes
    cdclW S S' and
    backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and
    get-all-levels-of-marked (trail S)
    = rev ([1.. $<(1 + \text{length (get-all-levels-of-marked (trail S))})$ ])) 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
    elim: conflictE propagateE)

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.. $< \text{Suc } (1 + \text{length (get-all-levels-of-marked ?M)})$ ]]
    unfolding k by simp
  finally show ?case using T undef by (auto simp add: defined-lit-map)
next
  case (backtrack L D K i M1 M2 T) note decomp = this(3) and confli = this(1) and T = this(7)
and
  all-marked = this(9) and bt-lvl = this(8)
  have atm-of L  $\notin$  atm-of ' lits-of-l M1
    using backtrack-lit-skipped[of S L K i M1 M2] backtrack(4,8-10) decomp
    by (fastforce simp add: lits-of-def)
  moreover then have undefined-lit M1 L
    by (auto simp: defined-lit-map lits-of-def)
  then have [simp]: trail T = Propagated L (mset-ccls D) # 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: rev-swap[symmetric] simp del: upt-simps)
  then show ?case

```

using T **by** (*auto simp: rev-swap M simp del: upt-simps dest!: append-cons-eq-upt(1)*)
qed *auto*

We write $1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ instead of $\text{backtrack-lvl } S$ to avoid non termination of rewriting.

definition $\text{cdcl}_W\text{-}M\text{-level-inv} :: 'st \Rightarrow \text{bool}$ **where**
 $\text{cdcl}_W\text{-}M\text{-level-inv } S \longleftrightarrow$
 $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$
 $\wedge \text{no-dup } (\text{trail } S)$
 $\wedge \text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$
 $\wedge \text{get-all-levels-of-marked } (\text{trail } S)$
 $= \text{rev } [1..<1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$

lemma $\text{cdcl}_W\text{-}M\text{-level-inv-decomp}$:
assumes $\text{cdcl}_W\text{-}M\text{-level-inv } S$
shows
 $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$ **and**
 $\text{no-dup } (\text{trail } S)$
using *assms* **unfolding** $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** *fastforce+*

lemma $\text{cdcl}_W\text{-consistent-inv}$:
fixes $S S' :: 'st$
assumes
 $\text{cdcl}_W S S'$ **and**
 $\text{cdcl}_W\text{-}M\text{-level-inv } S$
shows $\text{cdcl}_W\text{-}M\text{-level-inv } S'$
using *assms* $\text{cdcl}_W\text{-consistent-inv-2}$ $\text{cdcl}_W\text{-distinctinv-1}$ $\text{cdcl}_W\text{-bt}$ $\text{cdcl}_W\text{-bt-level'}$
unfolding $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** *meson+*

lemma $\text{rtrancpl-cdcl}_W\text{-consistent-inv}$:
assumes
 $\text{cdcl}_W^{**} S S'$ **and**
 $\text{cdcl}_W\text{-}M\text{-level-inv } S$
shows $\text{cdcl}_W\text{-}M\text{-level-inv } S'$
using *assms* **by** (*induct rule: rtrancpl-induct*) (*auto intro: cdcl_W-consistent-inv*)

lemma $\text{trancpl-cdcl}_W\text{-consistent-inv}$:
assumes
 $\text{cdcl}_W^{++} S S'$ **and**
 $\text{cdcl}_W\text{-}M\text{-level-inv } S$
shows $\text{cdcl}_W\text{-}M\text{-level-inv } S'$
using *assms* **by** (*induct rule: trancpl-induct*)
(*auto intro: cdcl_W-consistent-inv*)

lemma $\text{cdcl}_W\text{-}M\text{-level-inv-S0-cdcl}_W[\text{simp}]$:
 $\text{cdcl}_W\text{-}M\text{-level-inv } (\text{init-state } N)$
unfolding $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** *auto*

lemma $\text{cdcl}_W\text{-}M\text{-level-inv-get-level-le-backtrack-lvl}$:
assumes $\text{inv: cdcl}_W\text{-}M\text{-level-inv } S$
shows $\text{get-level } (\text{trail } S) L \leq \text{backtrack-lvl } S$

proof –
have $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [1..<1 + \text{backtrack-lvl } S]$
using *inv* **unfolding** $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** *auto*
then show *?thesis*


```

    using get-rev-level-less-max-get-all-levels-of-marked[of rev (trail S) 0 L]
    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  $\exists K\ M1\ M2. (Marked\ K\ (i + 1) \# M1, M2) \in set\ (get\_all\_marked\_decomposition\ (trail\ S))$ 
proof -
  let ?M = trail S
  have
    g:  $get\_all\_levels\_of\_marked\ (trail\ S) = rev\ [Suc\ 0..<Suc\ (backtrack\_lvl\ S)]$ 
    using M-l unfolding  $cdcl_W$ -M-level-inv-def by simp-all
  then have  $i+1 \in set\ (get\_all\_levels\_of\_marked\ (trail\ S))$ 
    using i-S by auto

  then obtain c K c' where tr-S:  $trail\ S = c @ Marked\ K\ (i + 1) \# c'$ 
    using in-get-all-levels-of-marked-iff-decomp[of i+1 trail S] by auto

  obtain M1 M2 where  $(Marked\ K\ (i + 1) \# M1, M2) \in set\ (get\_all\_marked\_decomposition\ (trail\ S))$ 
    using Marked-cons-in-get-all-marked-decomposition-append-Marked-cons unfolding tr-S by fast
  then show ?thesis by blast
qed

```

18.3.2 Better-Suited Induction Principle

We generalise the induction principle defined previously: the induction case for *backtrack* now includes the assumption that *undefined-lit* $M1\ L$. This helps the simplifier and thus the automation.

```

lemma backtrack-induction-lev[consumes 1, case-names M-devel-inv backtrack]:
  assumes
    bt: backtrack S T and
    inv:  $cdcl_W$ -M-level-inv S and
    backtrackH:  $\bigwedge K\ i\ M1\ M2\ L\ D\ T.
      raw\_conflicting\ S = Some\ D \implies
      L \in \# mset\_ccls\ D \implies
      (Marked\ K\ (Suc\ i) \# M1, M2) \in set\ (get\_all\_marked\_decomposition\ (trail\ S)) \implies
      get\_level\ (trail\ S)\ L = backtrack\_lvl\ S \implies
      get\_level\ (trail\ S)\ L = get\_maximum\_level\ (trail\ S)\ (mset\_ccls\ D) \implies
      get\_maximum\_level\ (trail\ S)\ (remove1\_mset\ L\ (mset\_ccls\ D)) \equiv i \implies
      undefined\_lit\ M1\ L \implies
      T \sim cons\_trail\ (Propagated\ L\ (cls\_of\_ccls\ D))
      (reduce\_trail\_to\ M1
        (add\_learned\_cls\ (cls\_of\_ccls\ D)
          (update\_backtrack\_lvl\ i
            (update\_conflicting\ None\ S)))) \implies
      P\ S\ T
  shows P S T
proof -
  obtain K i M1 M2 L D where
    decomp:  $(Marked\ K\ (Suc\ i) \# M1, M2) \in set\ (get\_all\_marked\_decomposition\ (trail\ S))$  and
    L:  $get\_level\ (trail\ S)\ L = backtrack\_lvl\ S$  and
    confl:  $raw\_conflicting\ S = Some\ D$  and
    LD:  $L \in \# mset\_ccls\ D$  and$ 
```

$lev-L$: $get-level (trail\ S)\ L = get-maximum-level (trail\ S)\ (mset-cls\ D)$ **and**
 $lev-D$: $get-maximum-level (trail\ S)\ (remove1-mset\ L\ (mset-cls\ D)) \equiv i$ **and**
 T : $T \sim cons-trail\ (Propagated\ L\ (cls-of-cls\ D))$
 $(reduce-trail-to\ M1$
 $(add-learned-cls\ (cls-of-cls\ D)$
 $(update-backtrack-lvl\ i$
 $(update-conflicting\ None\ S))))$
using bt **by** $(elim\ backtrackE)\ metis$

have $atm-of\ L \notin atm-of\ 'lits-of-l\ M1$
using $backtrack-lit-skipped[of\ S\ L\ K\ i\ M1\ M2]\ L\ decomp\ bt\ confl\ lev-L\ lev-D\ inv$
unfolding $cdcl_W-M-level-inv-def$ **by** $force$
then have $undefined-lit\ M1\ L$
by $(auto\ simp: defined-lit-map\ lits-of-def)$
then show $?thesis$
using $backtrackH[OF\ confl\ LD\ decomp\ L\ lev-L\ lev-D - T]$ **by** $simp$
qed

lemmas $backtrack-induction-lev2 = backtrack-induction-lev[consumes\ 2, case-names\ backtrack]$

lemma $cdcl_W-all-induct-lev-full$:
fixes $S :: 'st$
assumes
 $cdcl_W$: $cdcl_W\ S\ S'$ **and**
 $inv[simp]$: $cdcl_W-M-level-inv\ S$ **and**
 $propagateH$: $\bigwedge C\ L\ T. conflicting\ S = None \implies$
 $C !\in! raw-clauses\ S \implies$
 $L \in\# mset-cls\ C \implies$
 $trail\ S \models_{as}\ CNot\ (remove1-mset\ L\ (mset-cls\ C)) \implies$
 $undefined-lit\ (trail\ S)\ L \implies$
 $T \sim cons-trail\ (Propagated\ L\ C)\ S \implies$
 $P\ S\ T$ **and**
 $conflictH$: $\bigwedge D\ T. conflicting\ S = None \implies$
 $D !\in! raw-clauses\ S \implies$
 $trail\ S \models_{as}\ CNot\ (mset-cls\ D) \implies$
 $T \sim update-conflicting\ (Some\ (ccls-of-cls\ D))\ S \implies$
 $P\ S\ T$ **and**
 $forgetH$: $\bigwedge C\ T. conflicting\ S = None \implies$
 $C !\in! raw-learned-clss\ S \implies$
 $\neg(trail\ S) \models_{asm}\ clauses\ S \implies$
 $mset-cls\ C \notin set\ (get-all-mark-of-propagated\ (trail\ S)) \implies$
 $mset-cls\ C \notin\# init-clss\ S \implies$
 $T \sim remove-cls\ C\ S \implies$
 $P\ S\ T$ **and**
 $restartH$: $\bigwedge T. \neg trail\ S \models_{asm}\ clauses\ S \implies$
 $conflicting\ S = None \implies$
 $T \sim restart-state\ S \implies$
 $P\ S\ T$ **and**
 $decideH$: $\bigwedge L\ T. conflicting\ S = None \implies$
 $undefined-lit\ (trail\ S)\ L \implies$
 $atm-of\ L \in atms-of-mm\ (init-clss\ S) \implies$
 $T \sim cons-trail\ (Marked\ L\ (backtrack-lvl\ S + 1))\ (incr-lvl\ S) \implies$
 $P\ S\ T$ **and**
 $skipH$: $\bigwedge L\ C'\ M\ E\ T.$
 $trail\ S = Propagated\ L\ C'\ \# M \implies$

```

raw-conflicting S = Some E  $\implies$ 
-L  $\notin$  # mset-ccls E  $\implies$  mset-ccls E  $\neq$  {#}  $\implies$ 
T  $\sim$  tl-trail S  $\implies$ 
P S T and
resolveH:  $\bigwedge$  L E M D T.
trail S = Propagated L (mset-cls E) # M  $\implies$ 
L  $\in$  # mset-cls E  $\implies$ 
hd-raw-trail S = Propagated L E  $\implies$ 
raw-conflicting S = Some D  $\implies$ 
-L  $\in$  # mset-ccls D  $\implies$ 
get-maximum-level (trail S) (mset-ccls (remove-clit (-L) D)) = backtrack-lvl S  $\implies$ 
T  $\sim$  update-conflicting (Some (union-ccls (remove-clit (-L) D) (ccls-of-cls (remove-lit L E))))
(tl-trail S)  $\implies$ 
P S T and
backtrackH:  $\bigwedge$  K i M1 M2 L D T.
raw-conflicting S = Some D  $\implies$ 
L  $\in$  # mset-ccls D  $\implies$ 
(Marked K (Suc i) # M1, M2)  $\in$  set (get-all-marked-decomposition (trail S))  $\implies$ 
get-level (trail S) L = backtrack-lvl S  $\implies$ 
get-level (trail S) L = get-maximum-level (trail S) (mset-ccls D)  $\implies$ 
get-maximum-level (trail S) (remove1-mset L (mset-ccls D))  $\equiv$  i  $\implies$ 
undefined-lit M1 L  $\implies$ 
T  $\sim$  cons-trail (Propagated L (cls-of-ccls D))
(reduce-trail-to M1
(add-learned-cls (cls-of-ccls D)
(update-backtrack-lvl i
(update-conflicting None S))))  $\implies$ 
P S T
shows P S S'
using cdclW
proof (induct S' rule: cdclW-all-rules-induct)
case (propagate S')
then show ?case
by (auto elim!: propagateE intro!: propagateH)
next
case (conflict S')
then show ?case
by (auto elim!: conflictE intro!: conflictH)
next
case (restart S')
then show ?case
by (auto elim!: restartE intro!: restartH)
next
case (decide T)
then show ?case
by (auto elim!: decideE intro!: decideH)
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 by (auto elim!: forgetE intro!: forgetH)
next
  case (skip S')
  then show ?case by (auto elim!: skipE intro!: skipH)
next
  case (resolve S')
  then show ?case
    using hd-raw-trail[of S] by (cases trail S) (auto elim!: resolveE intro!: resolveH)
qed

lemmas cdclW-all-induct-lev2 = cdclW-all-induct-lev-full[consumes 2, case-names propagate conflict
forget restart decide skip resolve backtrack]

lemmas cdclW-all-induct-lev = cdclW-all-induct-lev-full[consumes 1, case-names lev-inv propagate
conflict forget restart decide skip resolve backtrack]

thm cdclW-o-induct
lemma cdclW-o-induct-lev[consumes 1, case-names M-lev decide skip resolve backtrack]:
fixes S :: 'st
assumes
  cdclW: cdclW-o S T and
  inv[simp]: cdclW-M-level-inv S and
  decideH:  $\bigwedge L$  T. conflicting S = None  $\implies$ 
    undefined-lit (trail S) L  $\implies$ 
    atm-of L  $\in$  atms-of-mm (init-cls S)  $\implies$ 
    T  $\sim$  cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S)  $\implies$ 
    P S T and
  skipH:  $\bigwedge L$  C' M E T.
    trail S = Propagated L C' # M  $\implies$ 
    raw-conflicting S = Some E  $\implies$ 
     $-L \notin \#$  mset-ccls E  $\implies$  mset-ccls E  $\neq$  {#}  $\implies$ 
    T  $\sim$  tl-trail S  $\implies$ 
    P S T and
  resolveH:  $\bigwedge L$  E M D T.
    trail S = Propagated L (mset-cls E) # M  $\implies$ 
    L  $\in \#$  mset-cls E  $\implies$ 
    hd-raw-trail S = Propagated L E  $\implies$ 
    raw-conflicting S = Some D  $\implies$ 
     $-L \in \#$  mset-ccls D  $\implies$ 
    get-maximum-level (trail S) (mset-ccls (remove-clit (-L) D)) = backtrack-lvl S  $\implies$ 
    T  $\sim$  update-conflicting (Some (union-ccls (remove-clit (-L) D) (ccls-of-cls (remove-lit L E))))
    (tl-trail S)  $\implies$ 
    P S T and
  backtrackH:  $\bigwedge K$  i M1 M2 L D T.
    raw-conflicting S = Some D  $\implies$ 
    L  $\in \#$  mset-ccls D  $\implies$ 
    (Marked K (Suc i) # M1, M2)  $\in$  set (get-all-marked-decomposition (trail S))  $\implies$ 
    get-level (trail S) L = backtrack-lvl S  $\implies$ 
    get-level (trail S) L = get-maximum-level (trail S) (mset-ccls D)  $\implies$ 
    get-maximum-level (trail S) (remove1-mset L (mset-ccls D))  $\equiv$  i  $\implies$ 
    undefined-lit M1 L  $\implies$ 
    T  $\sim$  cons-trail (Propagated L (cls-of-ccls D))
      (reduce-trail-to M1
        (add-learned-cls (cls-of-ccls D)
          (update-backtrack-lvl i

```

```

      (update-conflicting None S))))  $\impl$ 
    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 (auto elim!: decideE intro!: decideH)
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 (skip S')
  then show ?case by (auto elim!: skipE intro!: skipH)
next
  case (resolve S')
  then show ?case
    using hd-raw-trail[of S] by (cases trail S) (auto elim!: resolveE intro!: resolveH)
qed

lemmas cdclW-o-induct-lev2 = cdclW-o-induct-lev[consumes 2, case-names decide skip resolve
backtrack]

```

18.3.3 Compatibility with $op \sim$

```

lemma propagate-state-eq-compatible:
  assumes
    propa: propagate S T and
    SS':  $S \sim S'$  and
    TT':  $T \sim T'$ 
  shows propagate S' T'
proof -
  obtain C L where
    conf: conflicting S = None and
    C:  $C \notin$  raw-clauses S and
    L:  $L \in$  mset-cls C and
    tr: trail S  $\models_{as}$  CNot (remove1-mset L (mset-cls C)) and
    undef: undefined-lit (trail S) L and
    T:  $T \sim$  cons-trail (Propagated L C) S
  using propa by (elim propagateE) auto

  obtain C' where
    CC': mset-cls C' = mset-cls C and
    C':  $C' \notin$  raw-clauses S'
  using SS' C
  in-mset-clss-exists-preimage[of mset-cls C raw-learned-clss S']
  in-mset-clss-exists-preimage[of mset-cls C raw-init-clss S']
  apply -
  apply (frule in-clss-mset-clss)
  by (auto simp: state-eq-def raw-clauses-def simp del: state-simp dest: in-clss-mset-clss)

show ?thesis

```

apply (*rule propagate-rule*[*of - C*])
using *state-eq-sym*[*of S S'*] *SS' conf C' CC' L tr undef TT' T*
by (*auto simp: state-eq-def simp del: state-simp*)
qed

lemma *conflict-state-eq-compatible*:

assumes

conf: *conflict S T* **and**

TT': $T \sim T'$ **and**

SS': $S \sim S'$

shows *conflict S' T'*

proof –

obtain *D* **where**

conf: *conflicting S = None* **and**

D: $D \notin \text{raw-clauses } S$ **and**

tr: $\text{trail } S \models_{\text{as}} \text{CNot } (\text{mset-cls } D)$ **and**

T: $T \sim \text{update-conflicting } (\text{Some } (\text{ccls-of-cls } D)) \text{ } S$

using *conf* **by** (*elim conflictE*) *auto*

obtain *D'* **where**

DD': $\text{mset-cls } D' = \text{mset-cls } D$ **and**

D': $D' \notin \text{raw-clauses } S'$

using *D SS' in-mset-clss-exists-preimage* **by** *fastforce*

show *?thesis*

apply (*rule conflict-rule*[*of - D'*])

using *state-eq-sym*[*of S S'*] *SS' conf D' DD' tr TT' T*

by (*auto simp: state-eq-def simp del: state-simp*)

qed

lemma *backtrack-levE*[*consumes 2*]:

backtrack S S' \implies cdcl_W-M-level-inv S \implies

($\bigwedge K i M1 M2 L D$.

raw-conflicting S = Some D \implies

L $\in \# \text{mset-ccls } D \implies$

(Marked K (Suc i) $\# M1, M2$) $\in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \implies$

get-level (trail S) L = backtrack-lvl S \implies

get-level (trail S) L = get-maximum-level (trail S) (mset-ccls D) \implies

get-maximum-level (trail S) (remove1-mset L (mset-ccls D)) $\equiv i \implies$

undefined-lit M1 L \implies

S' $\sim \text{cons-trail } (\text{Propagated } L \text{ (cls-of-ccls } D))$

(reduce-trail-to M1

(add-learned-cls (cls-of-ccls D)

(update-backtrack-lvl i

(update-conflicting None S)))) $\implies P$ \implies

P

using *assms* **by** (*induction rule: backtrack-induction-lev2*) *metis*

thm *allI*

lemma *backtrack-state-eq-compatible*:

assumes

bt: *backtrack S T* **and**

SS': $S \sim S'$ **and**

TT': $T \sim T'$ **and**

inv: *cdcl_W-M-level-inv S*

shows *backtrack* $S' T'$
proof –
obtain $D L K i M1 M2$ **where**
conf: *raw-conflicting* $S = \text{Some } D$ **and**
L: $L \in \# \text{ mset-ccls } D$ **and**
decomp: $(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ **and**
lev: *get-level* $(\text{trail } S) L = \text{backtrack-lvl } S$ **and**
max: *get-level* $(\text{trail } S) L = \text{get-maximum-level } (\text{trail } S) (\text{mset-ccls } D)$ **and**
max-D: *get-maximum-level* $(\text{trail } S) (\text{remove1-mset } L (\text{mset-ccls } D)) \equiv i$ **and**
undef: *undefined-lit* $M1 L$ **and**
T: $T \sim \text{cons-trail } (\text{Propagated } L (\text{cls-of-ccls } D))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (\text{cls-of-ccls } D)$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting } \text{None } S))))$
using *bt inv* **by** $(\text{elim backtrack-levE})$ *metis*
obtain D' **where**
 D' : *raw-conflicting* $S' = \text{Some } D'$
using SS' *conf* **by** $(\text{cases raw-conflicting } S') \text{ auto}$
have $[simp]: \text{mset-ccls } D = \text{mset-ccls } D'$
using $SS' D' \text{ conf}$ **by** $(\text{auto simp: state-eq-def simp del: state-simp})[]$

have $T': T' \sim \text{cons-trail } (\text{Propagated } L (\text{cls-of-ccls } D'))$
 $(\text{reduce-trail-to } M1 (\text{add-learned-cls } (\text{cls-of-ccls } D')$
 $(\text{update-backtrack-lvl } i (\text{update-conflicting } \text{None } S'))))$
using TT' **unfolding** *state-eq-def*
using *decomp undef inv SS' T* **by** $(\text{auto simp add: cdcl}_W\text{-M-level-inv-def})$

show *?thesis*
apply $(\text{rule backtrack-rule}[\text{of } - D'])$
apply $(\text{rule } D')$
using *state-eq-sym* $[\text{of } S S'] TT' SS' D' \text{ conf } L \text{ decomp lev max max-D undef } T$
apply $(\text{auto simp: state-eq-def simp del: state-simp})[]$
using *decomp SS' lev SS' max-D max T'* **by** $(\text{auto simp: state-eq-def simp del: state-simp})$
qed

lemma *decide-state-eq-compatible*:
assumes
 $\text{decide } S T$ **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows *decide* $S' T'$
using *assms* **apply** (elim decideE)
apply $(\text{rule decide-rule})$
by $(\text{auto simp: state-eq-def raw-clauses-def simp del: state-simp})$

lemma *skip-state-eq-compatible*:
assumes
 $\text{skip: skip } S T$ **and**
 $SS': S \sim S'$ **and**
 $TT': T \sim T'$
shows *skip* $S' T'$
proof –
obtain $L C' M E$ **where**
 $\text{tr: trail } S = \text{Propagated } L C' \# M$ **and**

raw: raw-conflicting $S = \text{Some } E$ and
L: $-L \notin \# \text{ mset-ccls } E$ and
E: $\text{mset-ccls } E \neq \{\#\}$ and
T: $T \sim \text{tl-trail } S$
using *skip* **by** (*elim skipE*) *simp*
obtain E' **where** E' : raw-conflicting $S' = \text{Some } E'$
using SS' *raw* **by** (*cases raw-conflicting S'*) (*auto simp: state-eq-def simp del: state-simp*)
show ?thesis
apply (*rule skip-rule*)
using *tr raw L E T SS'* **apply** (*auto simp: simp del:*)[]
using E' **apply** *simp*
using $E' SS' L \text{ raw } E$ **apply** (*auto simp: state-eq-def simp del: state-simp*)[2]
using $T TT' SS'$ **by** (*auto simp: state-eq-def simp del: state-simp*)
qed

lemma *resolve-state-eq-compatible*:

assumes
res: *resolve* $S T$ and
TT': $T \sim T'$ and
SS': $S \sim S'$
shows *resolve* $S' T'$
proof –
obtain $E D L$ **where**
tr: *trail* $S \neq []$ and
hd: *hd-raw-trail* $S = \text{Propagated } L E$ and
L: $L \in \# \text{ mset-cls } E$ and
raw: raw-conflicting $S = \text{Some } D$ and
LD: $-L \in \# \text{ mset-ccls } D$ and
i: *get-maximum-level* (*trail* S) (*mset-ccls* (*remove-clit* ($-L$) D)) = *backtrack-lvl* S and
T: $T \sim \text{update-conflicting} (\text{Some} (\text{union-ccls} (\text{remove-clit} (-L) D) (\text{ccls-of-cls} (\text{remove-lit } L E)))) (\text{tl-trail } S)$
using *assms* **by** (*elim resolveE*) *simp*

obtain E' **where**
E': *hd-raw-trail* $S' = \text{Propagated } L E'$
using SS' *hd* **by** (*metis* (*trail S* $\neq []$) *hd-raw-trail is-proped-def marked-lit.disc*(3) *marked-lit.inject*(2) *mmset-of-mlit.elims state-eq-trail*)
have [*simp*]: *mset-cls* $E = \text{mset-cls } E'$
using *hd-raw-trail*[*of S*] *tr* *hd-raw-trail*[*of S'*] *tr SS'* *hd E'*
by (*metis* *marked-lit.inject*(2) *mmset-of-mlit.simps*(1) *state-eq-trail*)
obtain D' **where**
D': raw-conflicting $S' = \text{Some } D'$
using SS' *raw* **by** *fastforce*
have [*simp*]: *mset-ccls* $D = \text{mset-ccls } D'$
using $D' SS'$ *raw state-simp*(5) **by** *fastforce*
have $T'T$: $T' \sim T$
using TT' *state-eq-sym* **by** *auto*
show ?thesis
apply (*rule resolve-rule*)
using *tr SS'* **apply** *simp*
using E' **apply** *simp*
using L **apply** *simp*
using D' **apply** *simp*
using $D' SS'$ *raw LD* **apply** (*auto simp add: state-eq-def simp del: state-simp*)[]
using $D' SS'$ *raw LD* **apply** (*auto simp add: state-eq-def simp del: state-simp*)[]


```

    using raw SS' i apply (auto simp add: state-eq-def simp del: state-simp)[]
    using T T' T SS' by (auto simp: state-eq-def simp del: state-simp )
qed

```

lemma *forget-state-eq-compatible*:

assumes

forget: *forget S T* **and**

SS': $S \sim S'$ **and**

TT': $T \sim T'$

shows *forget S' T'*

proof –

obtain *C* **where**

conf: *conflicting S = None* **and**

C !∈! *raw-learned-clss S* **and**

tr: $\neg(\text{trail } S) \models_{asm} \text{clauses } S$ **and**

C1: *mset-cls C* \notin *set (get-all-mark-of-propagated (trail S))* **and**

C2: *mset-cls C* $\notin\#$ *init-clss S* **and**

T: $T \sim \text{remove-cls } C S$

using *forget* **by** (*elim forgetE*) *simp*

obtain *C'* **where**

C': *C' !∈! raw-learned-clss S'* **and**

[*simp*]: *mset-cls C' = mset-cls C*

using $\langle C !\in! \text{raw-learned-clss } S \rangle SS'$ *in-mset-clss-exists-preimage* **by** *fastforce*

show *?thesis*

apply (*rule forget-rule*)

using *SS' conf* **apply** *simp*

using *C'* **apply** *simp*

using *SS' tr* **apply** *simp*

using *SS' C1* **apply** *simp*

using *SS' C2* **apply** *simp*

using *T TT' SS'* **by** (*auto simp: state-eq-def simp del: state-simp*)

qed

lemma *cdcl_W-state-eq-compatible*:

assumes

cdcl_W S T **and** $\neg \text{restart } S T$ **and**

$S \sim S'$

$T \sim T'$ **and**

cdcl_W-M-level-inv S

shows *cdcl_W S' T'*

using *assms* **by** (*meson backtrack backtrack-state-eq-compatible bj cdcl_W.simps cdcl_W-o-rule-cases*

cdcl_W-rf.cases conflict-state-eq-compatible decide decide-state-eq-compatible forget

forget-state-eq-compatible propagate-state-eq-compatible resolve resolve-state-eq-compatible

skip skip-state-eq-compatible state-eq-ref)

lemma *cdcl_W-bj-state-eq-compatible*:

assumes

cdcl_W-bj S T **and** *cdcl_W-M-level-inv S*

$T \sim T'$

shows *cdcl_W-bj S T'*

using *assms* **by** (*meson backtrack backtrack-state-eq-compatible cdcl_W-bjE resolve*

resolve-state-eq-compatible skip skip-state-eq-compatible state-eq-ref)

18.3.4 Conservation of some Properties

lemma *cdcl_W-o-no-more-init-clss*:

assumes

cdcl_W-o S S' and

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss S'*

using *assms by (induct rule: cdcl_W-o-induct-lev2) (auto simp: inv cdcl_W-M-level-inv-decomp)*

lemma *trancpl-cdcl_W-o-no-more-init-clss*:

assumes

cdcl_W-o⁺⁺ S S' and

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss S'*

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

by *(auto dest: cdcl_W-o-no-more-init-clss*

dest!: trancpl-cdcl_W-consistent-inv dest: trancpl-mono-explicit[of cdcl_W-o - - cdcl_W]

simp: other)

lemma *rtrancpl-cdcl_W-o-no-more-init-clss*:

assumes

*cdcl_W-o^{**} S S' and*

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss S'*

using *assms unfolding rtrancpl-unfold by (auto intro: trancpl-cdcl_W-o-no-more-init-clss)*

lemma *cdcl_W-init-clss*:

assumes

cdcl_W S T and

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss T*

using *assms by (induct rule: cdcl_W-all-induct-lev2)*

(auto simp: inv cdcl_W-M-level-inv-decomp not-in-iff)

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*

18.3.5 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

$\wedge (\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{init-clss } S \models_{pm} T)$
 $\wedge \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \subseteq \text{set-mset } (\text{clauses } S)$

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(3) and confl = this(1) and undef = this(7)*
and *T = this(8)*

show *?case*

using *decomp confl learned undef T unfolding cdcl_W-learned-clause-def*

by (*auto dest!: get-all-marked-decomposition-exists-prepend*

simp: raw-clauses-def lev-inv cdcl_W-M-level-inv-decomp dest: true-clss-clss-left-right)

next

case (*resolve L C M D*) **note** *trail = this(1) and CL = this(2) and confl = this(4) and DL = this(5)*
and *lvl = this(6)*
and *T = this(7)*

moreover

have *init-clss S \models_{psm} learned-clss S*

using *learned trail unfolding cdcl_W-learned-clause-def raw-clauses-def* **by** *auto*

then have *init-clss S \models_{pm} mset-cl C + {#L#}*

using *trail learned unfolding cdcl_W-learned-clause-def raw-clauses-def*

by (*auto dest: true-clss-clss-in-imp-true-clss-clss*)

moreover have *remove1-mset (- L) (mset-ccls D) + {#- L#} = mset-ccls D*

using *DL* **by** (*auto simp: multiset-eq-iff*)

moreover have *remove1-mset L (mset-cl C) + {#L#} = mset-cl C*

using *CL* **by** (*auto simp: multiset-eq-iff*)

ultimately show *?case*

using *learned T*

by (*auto dest: mk-disjoint-insert*

simp add: cdcl_W-learned-clause-def raw-clauses-def

intro!: true-clss-clss-union-mset-true-clss-clss-or-not-true-clss-clss-or[of - L])

next

case (*restart T*)

then show *?case*

using *learned learned-clss-restart-state[of T]*

by (*auto*

simp: raw-clauses-def state-eq-def cdcl_W-learned-clause-def

simp del: state-simp

dest: true-clss-clssm-subsetE)

next

case *propagate*

then show *?case using learned* **by** (*auto simp: cdcl_W-learned-clause-def*)

next

case *conflict*

then show *?case using learned*

```

  by (fastforce simp: cdclW-learned-clause-def raw-clauses-def
      true-clss-clss-in-imp-true-clss-clss)
next
case (forget U)
then show ?case using learned
  by (auto simp: cdclW-learned-clause-def raw-clauses-def split: if-split-asm)
qed (auto simp: cdclW-learned-clause-def raw-clauses-def)

lemma rtrancp-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: rtrancp-cdclW-consistent-inv)

```

18.3.6 No alien atom in the state

This invariant means that all the literals are in the set of clauses.

```

definition no-strange-atm S'  $\longleftrightarrow$  (
  (∀ T. conflicting S' = Some T  $\longrightarrow$  atms-of T  $\subseteq$  atms-of-mm (init-clss S'))
  ∧ (∀ L mark. Propagated L mark ∈ set (trail S')
     $\longrightarrow$  atms-of (mark)  $\subseteq$  atms-of-mm (init-clss S'))
  ∧ atms-of-mm (learned-clss S')  $\subseteq$  atms-of-mm (init-clss S')
  ∧ atm-of ' (lits-of-l (trail S'))  $\subseteq$  atms-of-mm (init-clss S'))

```

```

lemma no-strange-atm-decomp:
  assumes no-strange-atm S
  shows conflicting S = Some T  $\implies$  atms-of T  $\subseteq$  atms-of-mm (init-clss S)
  and (∀ L mark. Propagated L mark ∈ set (trail S)
     $\longrightarrow$  atms-of (mark)  $\subseteq$  atms-of-mm (init-clss S))
  and atms-of-mm (learned-clss S)  $\subseteq$  atms-of-mm (init-clss S)
  and atm-of ' (lits-of-l (trail S))  $\subseteq$  atms-of-mm (init-clss S)
  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 in-atms-of-implies-atm-of-on-atms-of-ms:
  C + {#L#} ∈ # A  $\implies$  x ∈ atms-of C  $\implies$  x ∈ atms-of-mm A
  using multi-member-split by fastforce

```

```

lemma propagate-no-strange-atm-inv:
  assumes
    propagate S T and
    alien: no-strange-atm S
  shows no-strange-atm T
  using assms(1)
proof (induction)
  case (propagate-rule C L T) note confl = this(1) and C = this(2) and C-L = this(3) and
    tr = this(4) and undef = this(5) and T = this(6)
  have atm-CL: atms-of (mset-clss C)  $\subseteq$  atms-of-mm (init-clss S)
    using C alien unfolding no-strange-atm-def
    by (auto simp: raw-clauses-def atms-of-ms-def dest!: in-clss-mset-clss)
  show ?case

```

```

unfolding no-strange-atm-def
proof (intro conjI allI impI, goal-cases)
  case 1
  then show ?case
    using confl T undef by auto
next
  case (2 L' mark')
  then show ?case
    using C-L T alien undef atm-CL

    unfolding no-strange-atm-def raw-clauses-def apply auto by blast
next
  case (3)
  show ?case using T alien undef unfolding no-strange-atm-def by auto
next
  case (4)
  show ?case
    using T alien undef C-L atm-CL unfolding no-strange-atm-def by (auto simp: atms-of-def)
qed
qed

lemma in-atms-of-remove1-mset-in-atms-of:
   $x \in \text{atms-of } (\text{remove1-mset } L \ C) \implies x \in \text{atms-of } C$ 
  using in-diffD unfolding atms-of-def by fastforce

lemma cdclW-no-strange-atm-explicit:
  assumes
    cdclW S S' and
    lev: cdclW-M-level-inv S and
    conf:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S)$  and
    marked:  $\forall L \text{ mark}. \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S)$ 
       $\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S)$  and
    learned:  $\text{atms-of-mm } (\text{learned-clss } S) \subseteq \text{atms-of-mm } (\text{init-clss } S)$  and
    trail:  $\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{init-clss } S)$ 
  shows
     $(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S')) \wedge$ 
     $(\forall L \text{ mark}. \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S'))$ 
       $\longrightarrow \text{atms-of } ( \text{mark} ) \subseteq \text{atms-of-mm } (\text{init-clss } S')) \wedge$ 
     $\text{atms-of-mm } (\text{learned-clss } S') \subseteq \text{atms-of-mm } (\text{init-clss } S') \wedge$ 
     $\text{atm-of } '(\text{lits-of-l } (\text{trail } S')) \subseteq \text{atms-of-mm } (\text{init-clss } S')$ 
    (is ?C S'  $\wedge$  ?M S'  $\wedge$  ?U S'  $\wedge$  ?V S')
  using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
  case (propagate C L T) note confl = this(1) and C-L = this(2) and tr = this(3) and undef =
    this(4)
  and T = this(5)
  show ?case
    using propagate-rule[OF propagate.hyps(1-3) - propagate.hyps(5,6), simplified]
    propagate.hyps(4) propagate-no-strange-atm-inv[of S T]
    conf marked learned trail unfolding no-strange-atm-def by presburger
next
  case (decide L)
  then show ?case using learned marked conf trail unfolding raw-clauses-def by auto
next
  case (skip L C M D)

```

```

then show ?case using learned marked conf trail by auto
next
case (conflict D T) note D-S = this(2) and T = this(4)
have D: atm-of ' set-mset (mset-cls D)  $\subseteq$   $\bigcup$  (atms-of ' (set-mset (clauses S)))
  using D-S by (auto simp add: atms-of-def atms-of-ms-def)
moreover {
  fix xa :: 'v literal
  assume a1: atm-of ' set-mset (mset-cls D)  $\subseteq$  ( $\bigcup$   $x \in$  set-mset (init-clss S). atms-of x)
     $\cup$  ( $\bigcup$   $x \in$  set-mset (learned-clss S). atms-of x)
  assume a2:
    ( $\bigcup$   $x \in$  set-mset (learned-clss S). atms-of x)  $\subseteq$  ( $\bigcup$   $x \in$  set-mset (init-clss S). atms-of x)
  assume xa  $\in$  # mset-cls D
  then have atm-of xa  $\in$  UNION (set-mset (init-clss S)) atms-of
    using a2 a1 by (metis (no-types) Un-iff atm-of-lit-in-atms-of atms-of-def subset-Un-eq)
  then have  $\exists m \in$  set-mset (init-clss S). atm-of xa  $\in$  atms-of m
    by blast
} note H = this
ultimately show ?case using conflict.premis T learned marked conf trail
  unfolding atms-of-def atms-of-ms-def raw-clauses-def
  by (auto simp add: H)
next
case (restart T)
then show ?case using learned marked conf trail by auto
next
case (forget C T) note confl = this(1) and C = this(4) and C-le = 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-mm (init-clss S)
  using marked by simp
show ?case unfolding raw-clauses-def apply (intro conjI)
  using confl T trail C unfolding raw-clauses-def apply (auto dest!: H)[]
  using T trail C C-le apply (auto dest!: H)[]
  using T learned C-le atms-of-ms-remove-subset[of set-mset (learned-clss S)] apply auto[]
  using T trail C-le apply (auto simp: raw-clauses-def lits-of-def)[]
done
next
case (backtrack K i M1 M2 L D T) note confl = this(1) and LD = this(2) and decomp = this(3)
and
  undef = this(7)
  and T = this(8)
have ?C T
  using confl T decomp undef lev by (auto simp: cdclW-M-level-inv-decomp)
moreover have set M1  $\subseteq$  set (trail S)
  using decomp by auto
then have M: ?M T
  using marked conf undef confl T decomp lev
  by (auto simp: image-subset-iff raw-clauses-def cdclW-M-level-inv-decomp)
moreover have ?U T
  using learned decomp confl T undef lev unfolding raw-clauses-def
  by (auto simp: cdclW-M-level-inv-decomp)
moreover have ?V T
  using M confl confl trail T undef decomp lev LD
  by (auto simp: cdclW-M-level-inv-decomp atms-of-def
    dest!: get-all-marked-decomposition-exists-prepend)
ultimately show ?case by blast
next

```

```

case (resolve  $L\ C\ M\ D\ T$ ) note  $trail-S = this(1)$  and  $confl = this(4)$  and  $T = this(7)$ 
let  $?T = update-conflicting\ (Some\ ((remove-clit\ (-L)\ D)\ !\cup\ ccls-of-clis\ ((remove-lit\ L\ C))))$ 
  ( $tl-trail\ S$ )
have  $?C\ ?T$ 
  using  $confl\ trail-S\ conf\ marked\ by\ (auto\ dest!:\ in-atms-of-remove1-mset-in-atms-of)$ 
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  $simp$ 
qed

```

lemma $cdcl_W-no-strange-atm-inv$:
assumes $cdcl_W\ S\ S'$ **and** $no-strange-atm\ S$ **and** $cdcl_W-M-level-inv\ S$
shows $no-strange-atm\ S'$
using $cdcl_W-no-strange-atm-explicit[OF\ assms(1)]\ assms(2,3)$ **unfolding** $no-strange-atm-def$ **by** $fast$

lemma $rtrancpl-cdcl_W-no-strange-atm-inv$:
assumes $cdcl_W^{**}\ S\ S'$ **and** $no-strange-atm\ S$ **and** $cdcl_W-M-level-inv\ S$
shows $no-strange-atm\ S'$
using $assms$ **by** $induction\ (auto\ intro:\ cdcl_W-no-strange-atm-inv\ rtrancpl-cdcl_W-consistent-inv)$

18.3.7 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 $distinct-cdcl_W-state\ (S::'st)$
 $\longleftrightarrow ((\forall T.\ conflicting\ S = Some\ T \longrightarrow distinct-mset\ T)$
 $\wedge\ distinct-mset-mset\ (learned-clss\ S)$
 $\wedge\ distinct-mset-mset\ (init-clss\ S)$
 $\wedge\ (\forall L\ mark.\ (Propagated\ L\ mark \in set\ (trail\ S) \longrightarrow distinct-mset\ (mark))))$

lemma $distinct-cdcl_W-state-decomp$:
assumes $distinct-cdcl_W-state\ (S::'st)$
shows $\forall T.\ conflicting\ S = Some\ T \longrightarrow distinct-mset\ T$
and $distinct-mset-mset\ (learned-clss\ S)$
and $distinct-mset-mset\ (init-clss\ S)$
and $\forall L\ mark.\ (Propagated\ L\ mark \in set\ (trail\ S) \longrightarrow distinct-mset\ (mark))$
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 $conflicting\ S = Some\ T \implies distinct-mset\ T$
using $assms$ **unfolding** $distinct-cdcl_W-state-def$ **by** $auto$

lemma $distinct-cdcl_W-state-S0-cdcl_W[simp]$:
 $distinct-mset-mset\ (mset-clss\ N) \implies distinct-cdcl_W-state\ (init-state\ N)$
unfolding $distinct-cdcl_W-state-def$ **by** $auto$

lemma $distinct-cdcl_W-state-inv$:
assumes
 $cdcl_W\ S\ S'$ **and**
 $lev-inv:\ cdcl_W-M-level-inv\ S$ **and**

```

    distinct-cdclW-state S
  shows distinct-cdclW-state S'
  using assms(1,2,2,3)
proof (induct rule: cdclW-all-induct-lev2)
  case (backtrack L D K i M1 M2)
  then show ?case
    using lev-inv unfolding distinct-cdclW-state-def
    by (auto dest: get-all-marked-decomposition-incl simp: cdclW-M-level-inv-decomp)
next
  case restart
  then show ?case
    unfolding distinct-cdclW-state-def distinct-mset-set-def raw-clauses-def
    using learned-clss-restart-state[of S] by auto
next
  case resolve
  then show ?case
    by (auto simp add: distinct-cdclW-state-def distinct-mset-set-def raw-clauses-def
        distinct-mset-single-add
        intro!: distinct-mset-union-mset)
qed (auto simp: distinct-cdclW-state-def distinct-mset-set-def raw-clauses-def
    dest!: in-clss-mset-clss in-diffD)

lemma rtanclp-distinct-cdclW-state-inv:
  assumes
    cdclW** S S' and
    cdclW-M-level-inv S and
    distinct-cdclW-state S
  shows distinct-cdclW-state S'
  using assms apply (induct rule: rtanclp-induct)
  using distinct-cdclW-state-inv rtanclp-cdclW-consistent-inv by blast+

```

18.3.8 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* ⇒ bool **where**
every-mark-is-a-conflict *S* ≡
 $\forall L \text{ mark } a \text{ b. } a \text{ @ Propagated } L \text{ mark } \# b = (\text{trail } S) \longrightarrow (b \models_{\text{as}} \text{CNot } (\text{mark} - \{\#L\})) \wedge L \in \# \text{ mark}$

definition *cdcl_W-conflicting* *S* ≡
 $(\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T) \wedge \text{every-mark-is-a-conflict } S$

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 (trail *S*) (mset-ccls (remove-clit *L D*)) ≡ *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 (trail *S*) (mset-ccls *D*) and
S-conf: raw-conflicting *S* = Some *D* and
undef: undefined-lit *M1 L* and

$T: T \sim \text{cons-trail } (\text{Propagated } L \text{ (cls-of-ccls } D))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (\text{cls-of-ccls } D)$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting } \text{None } S)))) \text{ and}$
 $\text{confl: } \forall T. \text{ conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$
shows $\text{atms-of } (\text{mset-ccls } (\text{remove-clit } L \ D)) \subseteq \text{atm-of ' lits-of-l } (tl \ (\text{trail } T))$
proof $(\text{rule } ccontr)$
let $?k = \text{get-maximum-level } (\text{trail } S) \ (\text{mset-ccls } D)$
let $?D = \text{mset-ccls } D$
let $?D' = \text{mset-ccls } (\text{remove-clit } L \ D)$
have $\text{trail } S \models_{\text{as}} \text{CNot } ?D$ **using** $\text{confl } S\text{-confl}$ **by** auto
then have $\text{vars-of-}D: \text{atms-of } ?D \subseteq \text{atm-of ' lits-of-l } (\text{trail } S)$ **unfolding** atms-of-def
by $(\text{meson image-subsetI true-annots-CNot-all-atms-defined})$

obtain $M0$ **where** $M: \text{trail } S = M0 \ @ \ M2 \ @ \ \text{Marked } K \ (\text{Suc } i) \ \# \ M1$
using decomp **by** auto

have $\text{max: } ?k = \text{length } (\text{get-all-levels-of-marked } (M0 \ @ \ M2 \ @ \ \text{Marked } K \ (\text{Suc } i) \ \# \ M1))$
using $\text{inv unfolding cdcl}_W\text{-M-level-inv-def } S\text{-lvl } M$ **by** simp
assume $a: \neg ?thesis$
then obtain L' **where**
 $L': L' \in \text{atms-of } ?D' \text{ and}$
 $L'\text{-notin-}M1: L' \notin \text{atm-of ' lits-of-l } M1$
using $T \text{ undef decomp inv by (auto simp: cdcl}_W\text{-M-level-inv-decomp)}$
then have $L'\text{-in: } L' \in \text{atm-of ' lits-of-l } (M0 \ @ \ M2 \ @ \ \text{Marked } K \ (i + 1) \ \# \ [])$
using $\text{vars-of-}D \text{ unfolding } M$ **by** $(\text{auto dest: in-atms-of-remove1-mset-in-atms-of})$
then obtain L'' **where**
 $L'' \in \# \ ?D' \text{ and}$
 $L'': L' = \text{atm-of } L''$
using $L' \ L'\text{-notin-}M1$ **unfolding** atms-of-def **by** auto
have $\text{lev-}L'':$
 $\text{get-level } (\text{trail } S) \ L'' = \text{get-rev-level } (\text{Marked } K \ (\text{Suc } i) \ \# \ \text{rev } M2 \ @ \ \text{rev } M0) \ (\text{Suc } i) \ L''$
using $L'\text{-notin-}M1 \ L'' \ M$ **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 $\text{inv } S\text{-lvl unfolding cdcl}_W\text{-M-level-inv-def}$ **by** auto
then have $\text{get-all-levels-of-marked } (M0 \ @ \ M2) = \text{rev } [\text{Suc } (\text{Suc } i)..<\text{Suc } ?k]$
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 $\text{get-rev-level } (\text{Marked } K \ (\text{Suc } i) \ \# \ \text{rev } (M0 \ @ \ M2)) \ (\text{Suc } i) \ L''$
 $\geq \text{Min } (\text{set } ((\text{Suc } i) \ \# \ \text{get-all-levels-of-marked } (\text{Marked } K \ (\text{Suc } i) \ \# \ \text{rev } (M0 \ @ \ M2))))$
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 ... = Suc i by (auto intro: Min-eqI)
finally have get-rev-level (Marked K (Suc i) # rev (M0 @ M2)) (Suc i) L'' ≥ Suc i .
then have get-level (trail S) L'' ≥ i + 1
  using lev-L'' by simp
then have get-maximum-level (trail S) ?D' ≥ i + 1
  using get-maximum-level-ge-get-level[OF ‹L'' ∈# ?D'›, of trail S] by auto
then show False using i by auto
qed

```

lemma *distinct-atms-of-incl-not-in-other:*

```

assumes
  a1: no-dup (M @ M') and
  a2: atms-of D ⊆ atm-of ' lits-of-l M' and
  a3: x ∈ atms-of D
shows x ∉ atm-of ' lits-of-l M
proof -
  have ff1: ∧l ms. undefined-lit ms l ∨ atm-of l
    ∈ set (map (λm. atm-of (lit-of (m::('a, 'b, 'c) marked-lit))) ms)
    by (simp add: defined-lit-map)
  have ff2: ∧a. a ∉ atms-of D ∨ a ∈ atm-of ' lits-of-l M'
    using a2 by (meson subsetCE)
  have ff3: ∧a. a ∉ set (map (λm. atm-of (lit-of m)) M')
    ∨ a ∉ set (map (λm. atm-of (lit-of m)) M)
    using a1 by (metis (lifting) IntI distinct-append empty-iff map-append)
  have ∀ L a f. ∃ l. ((a::'a) ∉ f ' L ∨ (l::'a literal) ∈ L) ∧ (a ∉ f ' L ∨ f l = a)
    by blast
  then show x ∉ atm-of ' lits-of-l M
    using ff3 ff2 ff1 a3 by (metis (no-types) Marked-Propagated-in-iff-in-lits-of-l)
qed

```

lemma *true-annot-CNot-remove1-mset-remove1-mset:*

```

I ⊨as CNot C ⇒ I ⊨as CNot (remove1-mset L C)
by (auto simp: true-annots-true-clas-def-iff-negation-in-model dest: in-diffD)

```

lemma *cdcl_W-propagate-is-conclusion:*

```

assumes
  cdclW S S' and
  inv: cdclW-M-level-inv S and
  decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
  learned: cdclW-learned-clause S and
  confl: ∀ T. conflicting S = Some T ⇒ trail S ⊨as CNot T and
  alien: no-strange-atm S
shows all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
  case restart
  then show ?case by auto
next
  case forget
  then show ?case using decomp by auto
next
  case conflict
  then show ?case using decomp by auto
next

```

```

case (resolve  $L\ C\ M\ D$ ) note  $tr = this(1)$  and  $T = this(7)$ 
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  $L = this(3)$  and  $undef = this(5)$  and  $T = this(6)$ 
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  $unmark-l\ a\ \cup\ set-mset\ (init-clss\ S) \models_{ps} unmark-l\ y$ 
  using  $decomp\ ay$  unfolding all-decomposition-implies-def
  by (cases  $get-all-marked-decomposition\ (trail\ S)$ ) fastforce+
then have  $a-Un-N-M$ :  $unmark-l\ a\ \cup\ set-mset\ (init-clss\ S)$ 
   $\models_{ps} unmark-l\ (trail\ S)$ 
  unfolding  $M$  by (auto simp add: all-in-true-clss-clss image-Un)

have  $unmark-l\ a\ \cup\ set-mset\ (init-clss\ S) \models_p \{\#L\#\}$  (is  $?I \models_p -$ )
proof (rule true-clss-clss-plus-CNot)
  show  $?I \models_p remove1-mset\ L\ (mset-clss\ C) + \{\#L\#\}$ 
  apply (rule true-clss-clss-in-imp-true-clss-clss[of -
     $set-mset\ (init-clss\ S) \cup set-mset\ (learned-clss\ S)$ ])
  using  $learned\ propa\ L$  by (auto simp: raw-clauses-def cdclw-learned-clause-def
    true-annot-CNot-remove1-mset-remove1-mset)
next
have  $(\lambda m. \{\#lit-of\ m\#\})\ 'set\ (trail\ S) \models_{ps} CNot\ (remove1-mset\ L\ (mset-clss\ C))$ 
  using  $\langle (trail\ S) \models_{as} CNot\ (remove1-mset\ L\ (mset-clss\ C)) \rangle$  true-annots-true-clss-clss by blast
then show  $?I \models_{ps} CNot\ (remove1-mset\ L\ (mset-clss\ C))$ 
  using  $a-Un-N-M$  true-clss-clss-left-right true-clss-clss-union-l-r by blast
qed
moreover have  $\bigwedge aa\ b.$ 
   $\forall\ (Ls,\ seen) \in set\ (get-all-marked-decomposition\ (y\ @\ a)).$ 
   $unmark-l\ Ls\ \cup\ set-mset\ (init-clss\ S) \models_{ps} unmark-l\ seen$ 
   $\implies (aa,\ b) \in set\ (tl\ (get-all-marked-decomposition\ (y\ @\ a)))$ 
   $\implies unmark-l\ aa\ \cup\ set-mset\ (init-clss\ S) \models_{ps} unmark-l\ b$ 
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 ⟨unmark-l a ∪ set-mset (init-cls S) ⟦ps unmark-l y⟧
  ay by auto
next
case (backtrack K i M1 M2 L D T) note conf = this(1) and LD = this(2) and decomp' = this(3)
and
  lev-L = this(4) and undef = this(7) and T = this(8)
let ?D = mset-ccls D
let ?D' = mset-ccls (remove-clit L D)
have ∀ l ∈ set M2. ¬is-marked l
  using get-all-marked-decomposition-snd-not-marked decomp' 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 # M1))
    using T decomp' undef inv by (simp add: cdclW-M-level-inv-decomp)
  let ?m = get-all-marked-decomposition (Propagated L ?D # M1)
  let ?hd = hd ?m
  let ?tl = tl ?m
  consider
    (hd) x = ?hd
  | (tl) x ∈ set ?tl
  using x by (cases ?m) auto
  then show case x of (Ls, seen) ⇒ unmark-l Ls ∪ set-mset (init-cls T)
    ⟦ps unmark-l seen
  proof cases
    case 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 show ?thesis
      using decomp learned decomp confl alien inv T undef M
      unfolding all-decomposition-implies-def cdclW-M-level-inv-def
      by auto
  next
  case hd
  obtain M1' M1'' where M1: hd (get-all-marked-decomposition M1) = (M1', M1'')
    by (cases hd (get-all-marked-decomposition M1))
  then have x': x = (M1', Propagated L ?D # M1'')
    using ⟨x = ?hd⟩ by auto
  have (M1', M1'') ∈ set (get-all-marked-decomposition (trail S))
    using M1[symmetric] hd-get-all-marked-decomposition-skip-some[OF M1[symmetric],
      of M0 @ M2 - i+1] unfolding M by fastforce
  then have 1: unmark-l M1' ∪ set-mset (init-cls S) ⟦ps unmark-l M1''
    using decomp unfolding all-decomposition-implies-def by auto

  moreover
  have vars-of-D: atms-of ?D' ⊆ atm-of ' lits-of-l M1
    using backtrack-atms-of-D-in-M1[of S M1 L D i K M2 T] backtrack.hyps inv conf confl
    by (auto simp: cdclW-M-level-inv-decomp)
  have no-dup (trail S) using inv by (auto simp: cdclW-M-level-inv-decomp)

```

```

then have vars-in-M1:
   $\forall x \in \text{atms-of } ?D'. x \notin \text{atm-of } \text{'lits-of-l } (M0 @ M2 @ \text{Marked } K (i + 1) \# [])$ 
  using vars-of-D distinct-atms-of-incl-not-in-other[of
     $M0 @ M2 @ \text{Marked } K (i + 1) \# [] M1]$  unfolding M by auto
have trail S  $\models_{\text{as}} \text{CNot } (\text{remove1-mset } L (\text{mset-clss } D))$ 
  using conf confl LD unfolding M true-annots-true-clss-def-iff-negation-in-model
  by (auto dest!: Multiset.in-diffD)
then have  $M1 \models_{\text{as}} \text{CNot } ?D'$ 
  using vars-in-M1 true-annots-remove-if-notin-vars[of  $M0 @ M2 @ \text{Marked } K (i + 1) \# []$ 
     $M1 \text{ CNot } ?D']$  conf confl unfolding M lits-of-def by simp
have  $M1 = M1'' @ M1'$  by (simp add: M1 get-all-marked-decomposition-decomp)
have TT:  $\text{unmark-l } M1' \cup \text{set-mset } (\text{init-clss } S) \models_{\text{ps}} \text{CNot } ?D'$ 
  using true-annots-true-clss-clss[OF  $\langle M1 \models_{\text{as}} \text{CNot } ?D' \rangle$ ] true-clss-clss-left-right[OF 1]
  unfolding  $\langle M1 = M1'' @ M1' \rangle$  by (auto simp add: inf-sup-aci(5,7))
have  $\text{init-clss } S \models_{\text{pm}} ?D' + \{\#L\# \}$ 
  using conf learned confl LD unfolding cdclW-learned-clause-def by auto
then have  $T': \text{unmark-l } M1' \cup \text{set-mset } (\text{init-clss } S) \models_p ?D' + \{\#L\# \}$  by auto
have  $\text{atms-of } (?D' + \{\#L\# \}) \subseteq \text{atms-of-mm } (\text{clauses } S)$ 
  using alien conf LD unfolding no-strange-atm-def raw-clauses-def by auto
then have  $\text{unmark-l } M1' \cup \text{set-mset } (\text{init-clss } S) \models_p \{\#L\# \}$ 
  using true-clss-clss-plus-CNot[OF T' TT] by auto

ultimately show ?thesis
  using T' T decomp' undef inv unfolding x' by (simp add: cdclW-M-level-inv-decomp)
qed
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-clss S) (get-all-marked-decomposition (trail S)) and
  confl:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$  and
  alien: no-strange-atm S and
  mark-confl: every-mark-is-a-conflict S
shows every-mark-is-a-conflict S'
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
case (propagate C L T) note  $LC = \text{this}(3)$  and  $\text{confl} = \text{this}(4)$  and  $\text{undef} = \text{this}(5)$  and  $T = \text{this}(6)$ 
show ?case
  proof (intro allI impI)
    fix  $L' \text{ mark } a \text{ b}$ 
    assume  $a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } T$ 
    then consider
      (hd)  $a = []$  and  $L = L'$  and  $\text{mark} = \text{mset-clss } C$  and  $b = \text{trail } S$ 
      | (tl)  $tl \ a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } S$ 
    using T undef by (cases a) fastforce+
    then show  $b \models_{\text{as}} \text{CNot } (\text{mark} - \{\#L'\# \}) \wedge L' \in \# \text{ mark}$ 
    using mark-confl confl LC by cases auto
  qed
next
case (decide L) note  $\text{undef}[simp] = \text{this}(2)$  and  $T = \text{this}(4)$ 
have  $\bigwedge a \text{ La mark } b. a @ \text{Propagated } La \text{ mark } \# b = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S$ 

```

```

     $\Rightarrow$  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  $\forall La \text{ mark } a b. a @ \text{Propagated } La \text{ mark } \# b = \text{Propagated } L C' \# M$ 
       $\rightarrow b \models_{as} CNot (mark - \{\#La\}) \wedge La \in \# mark$ 
      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(7)
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 (mset-cls (L !++ C)) # a) @ Propagated L' mark # b
      = Propagated L (mset-cls (L !++ C)) # M
      using T tr-S by auto
    then show  $b \models_{as} CNot (mark - \{\#L'\}) \wedge L' \in \# mark$ 
      using mark-confl unfolding tr-S by (metis Cons-eq-appendI list.sel(3))
  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 conf = this(1) and LD = this(2) and decomp = this(3)
and
  undef = this(7) and T = this(8)
have  $\forall l \in \text{set } M2. \neg \text{is-marked } l$ 
  using get-all-marked-decomposition-snd-not-marked decomp by blast
obtain M0 where M: trail S = M0 @ M2 @ Marked K (i + 1) # M1
  using decomp by auto
have [simp]: trail (reduce-trail-to M1 (add-learned-cls (cls-of-ccls (insert-ccls L D))
  (update-backtrack-lvl i (update-conflicting None S)))) = M1
  using decomp lev by (auto simp: cdclW-M-level-inv-decomp)
let ?D = mset-ccls D
let ?D' = mset-ccls (remove-clit L D)
show ?case
  proof (intro allI impI)
    fix La :: 'v literal and mark :: 'v literal multiset and
      a b :: ('v, nat, 'v literal multiset) marked-lit list
    assume a @ Propagated La mark # b = trail T
    then consider

```

```

(hd-tr) a = [] and
  (Propagated La mark :: ('v, nat, 'v literal multiset) marked-lit)
    = Propagated L ?D and
    b = M1
| (tl-tr) tl a @ Propagated La mark # b = M1
using M T decomp undef lev by (cases a) (auto simp: cdclW-M-level-inv-def)
then show b ⊨as CNot (mark - {#La#}) ∧ La ∈# mark
proof cases
case hd-tr note A = this(1) and P = this(2) and b = this(3)
have trail S ⊨as CNot ?D using conf confl by auto
then have vars-of-D: atms-of ?D ⊆ atm-of ' lits-of-l (trail S)
  unfolding atms-of-def
  by (meson image-subsetI true-annots-CNot-all-atms-defined)
have vars-of-D: atms-of ?D' ⊆ atm-of ' lits-of-l M1
  using backtrack-atms-of-D-in-M1[of S M1 L D i K M2 T] T backtrack lev confl
  by (auto simp: cdclW-M-level-inv-decomp)
have no-dup (trail S) using lev by (auto simp: cdclW-M-level-inv-decomp)
then have ∀ x ∈ atms-of ?D'. x ∉ atm-of ' lits-of-l (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
then have M1 ⊨as CNot ?D'
  using true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # []
    M1 CNot ?D'] ⟨trail S ⊨as CNot ?D⟩ unfolding M lits-of-def
  by (simp add: true-annot-CNot-remove1-mset-remove1-mset)
then show b ⊨as CNot (mark - {#La#}) ∧ La ∈# mark
  using P LD b by auto
next
case tl-tr
then obtain c' where c' @ Propagated La mark # b = trail S
  unfolding M by auto
then show b ⊨as CNot (mark - {#La#}) ∧ La ∈# mark
  using mark-confl by auto
qed
qed
qed

```

lemma *cdcl_W-conflicting-is-false*:

```

assumes
  cdclW S S' and
  M-lev: cdclW-M-level-inv S and
  confl-inv: ∀ T. conflicting S = Some T ⟶ trail S ⊨as CNot T and
  marked-confl: ∀ L mark a b. a @ Propagated L mark # b = (trail S)
    ⟶ (b ⊨as CNot (mark - {#L#}) ∧ L ∈# mark) and
  dist: distinct-cdclW-state S
shows ∀ T. conflicting S' = Some T ⟶ trail S' ⊨as CNot T
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
case (skip L C' M D T) note tr-S = this(1) and confl = this(2) and L-D = this(3) and T = this(5)
let ?D = mset-ccls D
have D: Propagated L C' # M ⊨as CNot (mset-ccls D) using assms skip by auto
moreover
have L ∉# ?D
proof (rule ccontr)
assume ¬ ?thesis
then have - L ∈ lits-of-l M

```

```

    using in-CNot-implies-uminus(2)[of L ?D Propagated L C' # M]
    ⟨Propagated L C' # M ⊨as CNot ?D⟩ by simp
  then show False
    by (metis M-lev cdclW-M-level-inv-decomp(1) consistent-interp-def insert-iff
        lits-of-l-cons marked-lit.sel(2) skip.hyps(1))
  qed
ultimately show ?case
  using tr-S confl L-D T unfolding cdclW-M-level-inv-def
  by (auto intro: true-annots-CNot-lit-of-notin-skip)
next
  case (resolve L C M D T) note tr = this(1) and LC = this(2) and confl = this(4) and LD =
  this(5)
  and T = this(7)
  let ?C = remove1-mset L (mset-cls C)
  let ?D = remove1-mset (−L) (mset-ccls D)
  show ?case
  proof (intro allI impI)
    fix T'
    have tl (trail S) ⊨as CNot ?C using tr marked-confl by fastforce
  moreover
    have distinct-mset (?D + {#− L#}) using confl dist LD
      unfolding distinct-cdclW-state-def by auto
    then have −L ∉ # ?D unfolding distinct-mset-def
      by (meson ⟨distinct-mset (?D + {#− L#})⟩ distinct-mset-single-add)
    have M ⊨as CNot ?D
    proof −
      have Propagated L (?C + {#L#}) # M ⊨as CNot ?D ∪ CNot {#− L#}
        using confl tr confl-inv LC by (metis CNot-plus LD insert-DiffM2 option.simps(9))
      then show ?thesis
        using M-lev ⟨− L ∉ # ?D⟩ tr true-annots-lit-of-notin-skip
        unfolding cdclW-M-level-inv-def by force
    qed
  moreover assume conflicting T = Some T'
  ultimately
    show trail T ⊨as CNot T'
    using tr T by auto
  qed
qed (auto simp: M-lev cdclW-M-level-inv-decomp)

```

lemma *cdcl_W-conflicting-decomp*:

```

  assumes cdclW-conflicting S
  shows ∀ T. conflicting S = Some T ⟶ trail S ⊨as CNot T
  and ∀ L mark a b. a @ Propagated L mark # b = (trail S)
    ⟶ (b ⊨as CNot ( mark − {#L#}) ∧ L ∈ # mark)
  using assms unfolding cdclW-conflicting-def by blast+

```

lemma *cdcl_W-conflicting-decomp2*:

```

  assumes cdclW-conflicting S and conflicting S = Some T
  shows trail S ⊨as CNot T
  using assms unfolding cdclW-conflicting-def by blast+

```

lemma *cdcl_W-conflicting-S0-cdcl_W[simp]*:

```

  cdclW-conflicting (init-state N)
  unfolding cdclW-conflicting-def by auto

```


18.3.9 Putting all the invariants together

lemma *cdcl_W-all-inv*:

assumes

cdcl_W: *cdcl_W S S'* **and**

1: *all-decomposition-implies-m (init-clss S) (get-all-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'

proof –

show *S1: all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))*

using *cdcl_W-propagate-is-conclusion*[*OF cdcl_W 4 1 2 - 5*] 8 **unfolding** *cdcl_W-conflicting-def*
by *blast*

show *S2: cdcl_W-learned-clause S' using cdcl_W-learned-clss*[*OF cdcl_W 2 4*] .

show *S4: cdcl_W-M-level-inv S' using cdcl_W-consistent-inv*[*OF cdcl_W 4*] .

show *S5: no-strange-atm S' using cdcl_W-no-strange-atm-inv*[*OF cdcl_W 5 4*] .

show *S7: distinct-cdcl_W-state S' using distinct-cdcl_W-state-inv*[*OF cdcl_W 4 7*] .

show *S8: cdcl_W-conflicting S'*

using *cdcl_W-conflicting-is-false*[*OF cdcl_W 4 - - 7*] 8 *cdcl_W-propagate-is-false*[*OF cdcl_W 4 2 1 - 5*]

unfolding *cdcl_W-conflicting-def* **by** *fast*

qed

lemma *rtrancpl-cdcl_W-all-inv*:

assumes

cdcl_W: *rtrancpl 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: rtrancpl-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 (mset-clss 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
 $\forall T. \text{conflicting } (init-state N) = \text{Some } T \longrightarrow (\text{trail } (init-state N)) \models_{as} CNot T$ and
 no-strange-atm (init-state N) and
 consistent-interp (lits-of-l (trail (init-state N))) and
 $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# b = \text{trail } (init-state N) \longrightarrow$
 $(b \models_{as} CNot (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$ and
 distinct-cdcl_W-state (init-state N)
 using assms by auto

lemma cdcl_W-only-propagated-vars-unsat:
 assumes
 marked: $\forall x \in \text{set } M. \neg \text{is-marked } x$ and
 DN: $D \in \# \text{ clauses } S$ 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 $\neg \text{unsatisfiable } (set-mset N)$
 then obtain I where
 I: $I \models_s \text{set-mset } N$ and
 cons: consistent-interp I and
 tot: total-over-m I (set-mset N)
 unfolding satisfiable-def by auto
 have atms-of-mm $N \cup \text{atms-of-mm } U = \text{atms-of-mm } N$
 using atm-incl state unfolding total-over-m-def no-strange-atm-def
 by (auto simp add: raw-clauses-def)
 then have total-over-m I (set-mset N) using tot unfolding total-over-m-def by auto
 moreover then have total-over-m I (set-mset (learned-clss S))
 using atm-incl state unfolding no-strange-atm-def total-over-m-def total-over-set-def

```

  by auto
moreover have  $N \models_{psm} U$  using learned-cl state unfolding cdclW-learned-clause-def by auto
ultimately have  $I-D: I \models D$ 
  using  $I \ DN \ cons \ state \ unfolding \ true-clss-clss-def \ true-clss-def \ Ball-def$ 
  by (auto simp add: raw-clauses-def)

have  $l0: \{unmark \ L \mid L. \ is-marked \ L \wedge L \in set \ M\} = \{\}$  using marked by auto
have  $atms-of-ms \ (set-mset \ N \cup unmark-l \ M) = atms-of-mm \ N$ 
  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  $I \models_s unmark-l \ M$ 
  using all-decomposition-implies-propagated-lits-are-implied[OF inv] cons I
  unfolding true-clss-clss-def l0 by auto
then have  $IM: I \models_s unmark-l \ M$  by auto
{
  fix  $K$ 
  assume  $K \in \# \ D$ 
  then have  $-K \in lits-of-l \ M$ 
    using  $D \ unfolding \ true-annots-def \ Ball-def \ CNot-def \ true-annot-def \ true-clss-def \ true-lit-def$ 
    Bex-def by force
  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) $\implies ?N \cup \{unmark \ L \mid L. \ is-marked \ L \wedge L \in set \ ?M\} \models_{ps} unmark-l \ ?M$* , that show that the only choices we made are marked in the formula

```

lemma
  assumes all-decomposition-implies-m N (get-all-marked-decomposition M)
  and  $\forall m \in set \ M. \neg is-marked \ m$ 
  shows  $set-mset \ N \models_{ps} unmark-l \ M$ 
proof -
  have  $T: \{unmark \ L \mid L. \ is-marked \ L \wedge L \in set \ M\} = \{\}$  using assms(2) by auto
  then show ?thesis
    using all-decomposition-implies-propagated-lits-are-implied[OF assms(1)] unfolding T by simp
qed

```

lemma *conflict-with-false-implies-unsat*:

```

  assumes
     $cdcl_W: cdcl_W \ S \ S' \text{ and}$ 
     $lev: cdcl_W-M-level-inv \ S \text{ and}$ 
     $[simp]: conflicting \ S' = Some \ \{\#\} \text{ and}$ 
     $learned: cdcl_W-learned-clause \ S$ 
  shows unsatisfiable (set-mset (init-clss S))
  using assms
proof -
  have  $cdcl_W-learned-clause \ S' \text{ using } cdcl_W-learned-clss \ cdcl_W \ learned \ lev$  by auto
  then have  $init-clss \ S' \models_{pm} \{\#\}$  using assms(3) unfolding cdcl_W-learned-clause-def by auto
  then have  $init-clss \ S \models_{pm} \{\#\}$ 
    using  $cdcl_W-init-clss[OF assms(1) lev]$  by auto
  then show ?thesis unfolding satisfiable-def true-clss-clss-def by auto
qed

```

```

lemma conflict-with-false-implies-terminated:
  assumes  $cdcl_W \ S \ S'$ 
  and conflicting  $S = \text{Some } \{\#\}$ 
  shows False
  using assms by (induct rule: cdcl_W-all-induct) auto

```

18.3.10 No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

```

lemma learned-clss-are-not-tautologies:
  assumes
     $cdcl_W \ S \ S'$  and
    lev: cdcl_W-M-level-inv S and
    conflicting: cdcl_W-conflicting S and
    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  $K \ i \ M1 \ M2 \ L \ D \ T$ ) note confl = this(1)
have consistent-interp (lits-of-l (trail S)) using lev by (auto simp: cdcl_W-M-level-inv-decomp)
moreover
  have trail S  $\models_{as} CNot \ (mset-ccls \ D)$ 
    using conflicting confl unfolding cdcl_W-conflicting-def by auto
  then have lits-of-l (trail S)  $\models_s CNot \ (mset-ccls \ D)$ 
    using true-annots-true-cls by blast
  ultimately have  $\neg \text{tautology} \ (mset-ccls \ D)$  using consistent-CNot-not-tautology by blast
  then show ?case using backtrack no-tauto lev
    by (auto simp: cdcl_W-M-level-inv-decomp split: if-split-asm)
next
case restart
then show ?case using learned-clss-restart-state state-eq-learned-clss no-tauto
  by (metis (no-types, lifting) set-mset-mono subsetCE)
qed (auto dest!: in-diffD)

```

```

definition final-cdcl_W-state ( $S:: 'st$ )
 $\longleftrightarrow$  (trail S  $\models_{asm} \text{init-clss } S$ 
   $\vee ((\forall L \in \text{set} \ (trail \ S). \neg is\text{-marked } L) \wedge$ 
     $(\exists C \in \# \text{ init-clss } S. trail \ S \models_{as} CNot \ C)))$ 

```

```

definition termination-cdcl_W-state ( $S:: 'st$ )
 $\longleftrightarrow$  (trail S  $\models_{asm} \text{init-clss } S$ 
   $\vee ((\forall L \in \text{atms-of-mm} \ (\text{init-clss } S). L \in \text{atm-of } ' \text{ lits-of-l } (trail \ S))$ 
     $\wedge (\exists C \in \# \text{ init-clss } S. trail \ S \models_{as} CNot \ C)))$ 

```

18.4 CDCL Strong Completeness

```

fun mapi :: ( $'a \Rightarrow nat \Rightarrow 'b$ )  $\Rightarrow nat \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list}$  where
  mapi - - [] = [] |
  mapi  $f \ n \ (x \# xs) = f \ x \ n \ \# \ mapi \ f \ (n - 1) \ xs$ 

```

```

lemma mark-not-in-set-mapi[simp]:  $L \notin \text{set } M \Longrightarrow \text{Marked } L \ k \notin \text{set} \ (mapi \ \text{Marked } i \ M)$ 
  by (induct M arbitrary: i) auto

```

lemma *propagated-not-in-set-mapi*[simp]: $L \notin \text{set } M \implies \text{Propagated } L \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
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**
 $\text{atm-of ' (set } M) \subseteq \text{atms-of-mm (mset-clss } N)$
shows $\exists S. \text{rtrancp } \text{cdcl}_W \ (\text{init-state } N) \ S$
 $\wedge \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) \ M, \text{mset-clss } N, \{\#\}, \text{length } M, \text{None})$
using *assms*
proof (induct M)
case *Nil*
then show ?case **apply** – **by** (rule *exI*[of - *init-state* N]) auto
next
case (Cons $L \ M$) **note** $IH = \text{this}(1)$
have *consistent-interp* (set M) **and** *distinct* M **and** $\text{atm-of ' set } M \subseteq \text{atms-of-mm (mset-clss } N)$
using *Cons.prem*s(1–3) **unfolding** *consistent-interp-def* **by** auto
then obtain S **where**
 $st: \text{cdcl}_W^{**} \ (\text{init-state } N) \ S$ **and**
 $S: \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) \ M, \text{mset-clss } N, \{\#\}, \text{length } M, \text{None})$
using IH **by** *blast*
let $?S_0 = \text{incr-lvl } (\text{cons-trail } (\text{Marked } L \ (\text{length } M + 1)) \ S)$
have *undefined-lit* (mapi *Marked* (length M) M) L
using *Cons.prem*s(1,2) **unfolding** *defined-lit-def consistent-interp-def* **by** *fastforce*
moreover have *init-clss* $S = \text{mset-clss } N$
using S **by** *blast*
moreover have $\text{atm-of } L \in \text{atms-of-mm (mset-clss } N)$ **using** *Cons.prem*s(3) **by** auto
moreover have *undef: undefined-lit* (trail S) L
using $S \langle \text{distinct } (L \# M) \rangle$ *calculation*(1) **by** (auto simp: *defined-lit-mapi defined-lit-map*)
ultimately have $\text{cdcl}_W \ S \ ?S_0$
using $\text{cdcl}_W.\text{other}[OF \ \text{cdcl}_W.\text{o.decide}[OF \ \text{decide-rule}[of \ S \ L \ ?S_0]]] \ S$
by (auto simp: *state-eq-def simp del: state-simp*)
then have $\text{cdcl}_W^{**} \ (\text{init-state } N) \ ?S_0$
using st **by** auto
then show ?case
using $S \ \text{undef}$ **by** (auto intro!: *exI*[of - $?S_0$] *del: simp del:*)
qed

lemma *cdcl_W-strong-completeness*:
assumes
 $MN: \text{set } M \models_{sm} \text{mset-clss } N$ **and**
 $cons: \text{consistent-interp } (\text{set } M)$ **and**
 $dist: \text{distinct } M$ **and**

$atm: atm\text{-}of \text{ ` } (set\ M) \subseteq atm\text{-}of\text{-}mm\ (mset\text{-}class\ N)$
obtains S **where**
 $state\ S = (mapi\ Marked\ (length\ M)\ M, mset\text{-}class\ N, \{\#\}, length\ M, None)$ **and**
 $rtranclp\ cdcl_W\ (init\text{-}state\ N)\ S$ **and**
 $final\text{-}cdcl_W\text{-}state\ S$
proof –
obtain S **where**
 $st: rtranclp\ cdcl_W\ (init\text{-}state\ N)\ S$ **and**
 $S: state\ S = (mapi\ Marked\ (length\ M)\ M, mset\text{-}class\ N, \{\#\}, length\ M, None)$
using $cdcl_W\text{-}can\text{-}do\text{-}step[OF\ cons\ dist\ atm]$ **by** $auto$
have $lits\text{-}of\text{-}l\ (mapi\ Marked\ (length\ M)\ M) = set\ M$
by $(induct\ M, auto)$
then have $mapi\ Marked\ (length\ M)\ M \models_{asm}\ mset\text{-}class\ N$ **using** $MN\ true\text{-}annot\text{-}true\text{-}cls$ **by** $metis$
then have $final\text{-}cdcl_W\text{-}state\ S$
using $S\ unfolding\ final\text{-}cdcl_W\text{-}state\text{-}def$ **by** $auto$
then show $?thesis$ **using** $that\ st\ S$ **by** $blast$
qed

18.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

18.5.1 Definition

lemma $tranclp\text{-}conflict$:
 $tranclp\ conflict\ S\ S' \implies conflict\ S\ S'$
apply $(induct\ rule: tranclp.induct)$
apply $simp$
by $(metis\ conflictE\ conflicting\text{-}update\text{-}conflicting\ option.distinct(1)\ option.simps(8,9)\ state\text{-}eq\text{-}conflicting)$

lemma $tranclp\text{-}conflict\text{-}iff[iff]$:
 $full1\ conflict\ S\ S' \longleftrightarrow conflict\ S\ S'$
proof –
have $tranclp\ conflict\ S\ S' \implies conflict\ S\ S'$ **by** $(meson\ tranclp\text{-}conflict\ rtranclpD)$
then show $?thesis$ **unfolding** $full1\text{-}def$
by $(metis\ conflict.simps\ conflicting\text{-}update\text{-}conflicting\ option.distinct(1)\ option.simps(9)\ state\text{-}eq\text{-}conflicting\ tranclp.intros(1))$
qed

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

lemma $rtranclp\text{-}cdcl_W\text{-}cp\text{-}rtranclp\text{-}cdcl_W$:
 $cdcl_W\text{-}cp^{**}\ S\ T \implies cdcl_W^{**}\ S\ T$
by $(induction\ rule: rtranclp\text{-}induct)\ (auto\ simp: cdcl_W\text{-}cp.simps\ dest: cdcl_W.intros)$

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-cdclW-cp-state-eq-compatible:
  assumes
    cdclW-cp++ S T and
    S ~ S' and
    T ~ T'
  shows cdclW-cp++ S' T'
  using assms
proof induction
  case base
  then show ?case
    using cdclW-cp-state-eq-compatible by blast
next
  case (step U V)
  obtain ss :: 'st where
    cdclW-cp S ss ∧ cdclW-cp** ss U
  by (metis (no-types) step(1) trancplD)
  then show ?case
    by (meson cdclW-cp-state-eq-compatible rtrancpl.rtrancpl-into-rtrancpl rtrancpl-into-trancpl2
      state-eq-ref step(2) step(4) step(5))
qed

lemma option-full-cdclW-cp:
  conflicting S ≠ None ⇒ full cdclW-cp S S
  unfolding full-def rtrancpl-unfold trancpl-unfold
  by (auto simp add: cdclW-cp.simps elim: conflictE propagateE)

lemma skip-unique:
  skip S T ⇒ skip S T' ⇒ T ~ T'
  by (fastforce simp: state-eq-def simp del: state-simp elim: skipE)

lemma resolve-unique:
  resolve S T ⇒ resolve S T' ⇒ T ~ T'
  by (fastforce simp: state-eq-def simp del: state-simp elim: resolveE)

lemma cdclW-cp-no-more-clauses:
  assumes cdclW-cp S S'
  shows clauses S = clauses S'
  using assms by (induct rule: cdclW-cp.induct) (auto elim!: conflictE propagateE)

lemma trancpl-cdclW-cp-no-more-clauses:
  assumes cdclW-cp++ S S'
  shows clauses S = clauses S'
  using assms by (induct rule: trancpl.induct) (auto dest: cdclW-cp-no-more-clauses)

lemma rtrancpl-cdclW-cp-no-more-clauses:
  assumes cdclW-cp** S S'
  shows clauses S = clauses S'
  using assms by (induct rule: rtrancpl-induct) (fastforce dest: cdclW-cp-no-more-clauses)+

lemma no-conflict-after-conflict:
  conflict S T ⇒ ¬conflict T U

```

by (metis None-eq-map-option-iff conflictE conflicting-update-conflicting option.distinct(1)
state-simp(5))

lemma no-propagate-after-conflict:

conflict S T $\implies \neg$ propagate T U

by (metis conflictE conflicting-update-conflicting map-option-is-None option.distinct(1)
propagate.cases state-eq-conflicting)

lemma tranclp-cdcl_W-cp-propagate-with-conflict-or-not:

assumes cdcl_W-cp⁺⁺ S U

shows (propagate⁺⁺ S U \wedge conflicting U = None)

$\vee (\exists T D. \text{propagate}^{**} S T \wedge \text{conflict} T U \wedge \text{conflicting} U = \text{Some } D)$

proof –

have 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 tranclp-into-rtranclp dest: no-conflict-after-conflict
no-propagate-after-conflict)+

moreover

have propagate⁺⁺ S U \implies conflicting U = None

unfolding tranclp-unfold-end by (auto elim!: propagateE)

moreover

have $\bigwedge T. \text{conflict} T U \implies \exists D. \text{conflicting} U = \text{Some } D$

by (auto elim!: conflictE simp: state-eq-def simp del: state-simp)

ultimately show ?thesis by meson

qed

lemma cdcl_W-cp-conflicting-not-empty[simp]: conflicting S = Some D $\implies \neg$ cdcl_W-cp S S'

proof

assume cdcl_W-cp S S' and conflicting S = Some D

then show False by (induct rule: cdcl_W-cp.induct)

(auto elim: conflictE propagateE simp: state-eq-def simp del: state-simp)

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 cdcl_W-o S S' and re-apply conflict and propagate full cdcl_W-cp S' S''

inductive cdcl_W-stgy :: 'st \Rightarrow 'st \Rightarrow bool for S :: 'st where

conflict': full1 cdcl_W-cp S S' \implies cdcl_W-stgy S S' |

other': cdcl_W-o S S' \implies no-step cdcl_W-cp S \implies full cdcl_W-cp S' S'' \implies cdcl_W-stgy S S''

18.5.2 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 = learned-clss S'

using assms by (induct rule: cdcl_W-cp.induct) (fastforce elim: conflictE propagateE)+

lemma rtranclp-cdcl_W-cp-learned-clause-inv:

assumes $cdcl_W\text{-}cp^{**} S S'$
shows $learned\text{-}clss S = learned\text{-}clss S'$
using *assms* **by** (*induct rule*: $rtrancpl\text{-}induct$) (*fastforce dest*: $cdcl_W\text{-}cp\text{-}learned\text{-}clause\text{-}inv$)+

lemma $trancpl\text{-}cdcl_W\text{-}cp\text{-}learned\text{-}clause\text{-}inv$:
assumes $cdcl_W\text{-}cp^{++} S S'$
shows $learned\text{-}clss S = learned\text{-}clss S'$
using *assms* **by** (*simp add*: $rtrancpl\text{-}cdcl_W\text{-}cp\text{-}learned\text{-}clause\text{-}inv$ $trancpl\text{-}into\text{-}rtrancpl$)

lemma $cdcl_W\text{-}cp\text{-}backtrack\text{-}lvl$:
assumes $cdcl_W\text{-}cp S S'$
shows $backtrack\text{-}lvl S = backtrack\text{-}lvl S'$
using *assms* **by** (*induct rule*: $cdcl_W\text{-}cp.induct$) (*fastforce elim*: $conflictE$ $propagateE$)+

lemma $rtrancpl\text{-}cdcl_W\text{-}cp\text{-}backtrack\text{-}lvl$:
assumes $cdcl_W\text{-}cp^{**} S S'$
shows $backtrack\text{-}lvl S = backtrack\text{-}lvl S'$
using *assms* **by** (*induct rule*: $rtrancpl\text{-}induct$) (*fastforce dest*: $cdcl_W\text{-}cp\text{-}backtrack\text{-}lvl$)+

lemma $cdcl_W\text{-}cp\text{-}consistent\text{-}inv$:
assumes $cdcl_W\text{-}cp S S'$
and $cdcl_W\text{-}M\text{-}level\text{-}inv S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv S'$
using *assms*
proof (*induct rule*: $cdcl_W\text{-}cp.induct$)
case ($conflict'$)
then show ?*case* **using** $cdcl_W\text{-}consistent\text{-}inv$ $cdcl_W.conflict$ **by** *blast*
next
case ($propagate' S S'$)
have $cdcl_W S S'$
using $propagate'.hypos(1)$ $propagate$ **by** *blast*
then show $cdcl_W\text{-}M\text{-}level\text{-}inv S'$
using $propagate'.prems(1)$ $cdcl_W\text{-}consistent\text{-}inv$ $propagate$ **by** *blast*
qed

lemma $full1\text{-}cdcl_W\text{-}cp\text{-}consistent\text{-}inv$:
assumes $full1\text{-}cdcl_W\text{-}cp S S'$
and $cdcl_W\text{-}M\text{-}level\text{-}inv S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv S'$
using *assms* **unfolding** $full1\text{-}def$
by (*metis* $rtrancpl\text{-}cdcl_W\text{-}cp\text{-}rtrancpl\text{-}cdcl_W$ $rtrancpl\text{-}unfold$ $trancpl\text{-}cdcl_W\text{-}consistent\text{-}inv$)

lemma $rtrancpl\text{-}cdcl_W\text{-}cp\text{-}consistent\text{-}inv$:
assumes $rtrancpl\text{-}cdcl_W\text{-}cp S S'$
and $cdcl_W\text{-}M\text{-}level\text{-}inv S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv S'$
using *assms* **unfolding** $full1\text{-}def$
by (*induction rule*: $rtrancpl\text{-}induct$) (*blast intro*: $cdcl_W\text{-}cp\text{-}consistent\text{-}inv$)+

lemma $cdcl_W\text{-}stgy\text{-}consistent\text{-}inv$:
assumes $cdcl_W\text{-}stgy S S'$
and $cdcl_W\text{-}M\text{-}level\text{-}inv S$
shows $cdcl_W\text{-}M\text{-}level\text{-}inv S'$
using *assms* **apply** (*induct rule*: $cdcl_W\text{-}stgy.induct$)
unfolding $full\text{-}unfold$ **by** (*blast intro*: $cdcl_W\text{-}consistent\text{-}inv$ $full1\text{-}cdcl_W\text{-}cp\text{-}consistent\text{-}inv$)

$cdcl_W.other)+$

lemma *rtrancp-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 elim*: $conflictE propagateE$)

lemma *trancp-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*: $trancp.induct$) (*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*: $trancp-cdcl_W-cp-no-more-init-clss$
 $trancp-cdcl_W-o-no-more-init-clss$)

by (*metis* $cdcl_W-o-no-more-init-clss$ $rtrancp-unfold$ $trancp-cdcl_W-cp-no-more-init-clss$)

lemma *rtrancp-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*: $rtrancp-induct$, *simp*)

using $cdcl_W-stgy-no-more-init-clss$ **by** (*simp add*: $rtrancp-cdcl_W-stgy-consistent-inv$)

lemma *cdcl_W-cp-dropWhile-trail'*:

assumes $cdcl_W-cp S S'$

obtains M **where** $trail S' = M @ trail S$ **and** $(\forall l \in set M. \neg is-marked l)$

using *assms* **by** *induction* (*fastforce elim*: $conflictE propagateE$) $+$

lemma *rtrancp-cdcl_W-cp-dropWhile-trail'*:

assumes $cdcl_W-cp^{**} S S'$

obtains $M :: ('v, nat, 'v\ clause) marked-lit list$ **where**

$trail S' = M @ trail S$ **and** $\forall l \in set M. \neg is-marked l$

using *assms* **by** *induction* (*fastforce dest!*: $cdcl_W-cp-dropWhile-trail'$) $+$

lemma *cdcl_W-cp-dropWhile-trail*:

assumes $cdcl_W-cp S S'$

shows $\exists M. trail S' = M @ trail S \wedge (\forall l \in set M. \neg is-marked l)$

using *assms* **by** *induction* (*fastforce elim*: $conflictE propagateE$) $+$

lemma *rtrancp-cdcl_W-cp-dropWhile-trail*:

assumes $cdcl_W-cp^{**} S S'$

shows $\exists M. trail S' = M @ trail S \wedge (\forall l \in set M. \neg is-marked l)$

using *assms* **by** *induction* (*fastforce dest*: $cdcl_W-cp-dropWhile-trail$) $+$

This theorem can be seen as a termination theorem for $cdcl_W-cp$.

lemma *length-model-le-vars*:

assumes

no-strange-atm *S* **and**
no-d: *no-dup* (*trail S*) **and**
finite (*atms-of-mm* (*init-clss S*))

shows $\text{length } (\text{trail } S) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } S))$

proof –

obtain *M N U k D* **where** *S*: *state S = (M, N, U, k, D)* **by** (*cases state S, auto*)

have *finite* (*atm-of* ‘*lits-of-l*’ (*trail S*))

using *assms*(1,3) **unfolding** *S* **by** (*auto simp add: finite-subset*)

have $\text{length } (\text{trail } S) = \text{card } (\text{atm-of } \text{‘lits-of-l’ } (\text{trail } S))$

using *no-dup-length-eq-card-atm-of-lits-of-l no-d* **by** *blast*

then show *?thesis* **using** *assms*(1) **unfolding** *no-strange-atm-def*

by (*auto simp add: assms*(3) *card-mono*)

qed

lemma *cdcl_W-cp-decreasing-measure*:

assumes

cdcl_W: *cdcl_W-cp S T* **and**
M-lev: *cdcl_W-M-level-inv S* **and**
alien: *no-strange-atm S*

shows $(\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$

$+ (\text{if } \text{conflicting } S = \text{None then } 1 \text{ else } 0)) S$

$> (\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$

$+ (\text{if } \text{conflicting } S = \text{None then } 1 \text{ else } 0)) T$

using *assms*

proof –

have $\text{length } (\text{trail } T) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } T))$

apply (*rule length-model-le-vars*)

using *cdcl_W-no-strange-atm-inv alien M-lev* **apply** (*meson cdcl_W cdcl_W.simps cdcl_W-cp.cases*)

using *M-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 elim!: conflictE propagateE*

simp del: state-simp simp: state-eq-def)+

qed

lemma *cdcl_W-cp-wf*: *wf {(b,a). (cdcl_W-M-level-inv a ∧ no-strange-atm a)*

∧ cdcl_W-cp a b}

apply (*rule wf-wf-if-measure* [of *less-than - -*

$(\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$

$+ (\text{if } \text{conflicting } S = \text{None then } 1 \text{ else } 0))]$)

apply *simp*

using *cdcl_W-cp-decreasing-measure* **unfolding** *less-than-iff* **by** *blast*

lemma *rtrancp-cdcl_W-all-struct-inv-cdcl_W-cp-iff-rtrancp-cdcl_W-cp*:

assumes

lev: *cdcl_W-M-level-inv S* **and**

alien: *no-strange-atm S*

shows $(\lambda a b. (\text{cdcl}_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge \text{cdcl}_W\text{-cp } a b)^{**} S T$

$\longleftrightarrow \text{cdcl}_W\text{-cp}^{**} S T$

(*is ?I S T* \longleftrightarrow *?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: assms(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 assms unfolding full-def
    by blast
  moreover
  then have cdclW** S T
    using rtrancpl-cdclW-cp-rtrancpl-cdclW by blast
  then have
    cdclW-M-level-inv T and
    no-strange-atm T
    using ⟨cdclW** S T⟩ apply (simp add: assms(1) rtrancpl-cdclW-consistent-inv)
    using ⟨cdclW** S T⟩ assms(2) rtrancpl-cdclW-no-strange-atm-inv lev by blast
  then have no-step cdclW-cp T
    using T unfolding full-def by auto
  ultimately show thesis using that unfolding full-def by blast
qed

lemma always-exists-full-cdclW-cp-step:
  assumes no-strange-atm S
  shows ∃ S''. full cdclW-cp S S''
  using assms

```

```

proof (induct card (atms-of-mm (init-clss S) - atm-of 'lits-of-l (trail S)) arbitrary: S)
  case 0 note card = this(1) and alien = this(2)
  then have atm: atms-of-mm (init-clss S) = atm-of 'lits-of-l (trail S)
    unfolding no-strange-atm-def by auto
  { 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 simp: cdclW-cp.simps elim!: conflictE propagateE
        simp del: state-simp simp: state-eq-def)
    then have ?case using a S' cdclW-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 E L where
    S: conflicting S = None and
    E: E ! $\in$ ! raw-clauses S and
    LE: L  $\in$  # mset-cls E and
    tr: trail S  $\models_{\text{as}}$  CNot (mset-cls (remove-lit L E)) and
    undef: undefined-lit (trail S) L and
    S': S'  $\sim$  cons-trail (Propagated L E) S
    by (elim propagateE) simp
  have atms-of-mm (learned-clss S)  $\subseteq$  atms-of-mm (init-clss S)
    using alien S unfolding no-strange-atm-def by auto
  then have atm-of L  $\in$  atms-of-mm (init-clss S)
    using E LE S undef unfolding raw-clauses-def by (force simp: in-implies-atm-of-on-atms-of-ms)
  then have False using undef S unfolding atm unfolding lits-of-def
    by (auto simp add: defined-lit-map)
  }
  ultimately show ?case unfolding full-def by (metis cdclW-cp.cases 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 simp: cdclW-cp.simps elim!: conflictE propagateE
        simp del: state-simp simp: state-eq-def)
    then have ?case unfolding full-def Ex-def using S' cdclW-cp.conflict' by blast
  }
moreover {
  assume a:  $\exists S'. \text{propagate } S S'$ 
  then obtain S' where propagate: propagate S S' by blast
  then obtain E L where
    S: conflicting S = None and
    E: E ! $\in$ ! raw-clauses S and
    LE: L  $\in$  # mset-cls E and
    tr: trail S  $\models_{\text{as}}$  CNot (mset-cls (remove-lit L E)) and
    undef: undefined-lit (trail S) L and
    S': S'  $\sim$  cons-trail (Propagated L E) S
    by (elim propagateE) simp
  then have atm-of L  $\notin$  atm-of 'lits-of-l (trail S)
    unfolding lits-of-def by (auto simp add: defined-lit-map)
  moreover
    have no-strange-atm S' using alien propagate propagate-no-strange-atm-inv by blast
    then have atm-of L  $\in$  atms-of-mm (init-clss S)
  }

```

```

    using S' LE E undef unfolding no-strange-atm-def
    by (auto simp: raw-clauses-def in-implies-atm-of-on-atms-of-ms)
    then have  $\bigwedge A. \{ \text{atm-of } L \} \subseteq \text{atms-of-mm } (\text{init-clss } S) - A \vee \text{atm-of } L \in A$  by force
    moreover have  $\text{Suc } n - \text{card } \{ \text{atm-of } L \} = n$  by simp
    moreover have  $\text{card } (\text{atms-of-mm } (\text{init-clss } S) - \text{atm-of } \text{' lits-of-l } (\text{trail } S)) = \text{Suc } n$ 
    using card S S' by simp
    ultimately
    have  $\text{card } (\text{atms-of-mm } (\text{init-clss } S) - \text{atm-of } \text{' insert } L (\text{lits-of-l } (\text{trail } S))) = n$ 
    by (metis (no-types) Diff-insert card-Diff-subset finite.emptyI finite.insertI image-insert)
    then have  $n = \text{card } (\text{atms-of-mm } (\text{init-clss } S') - \text{atm-of } \text{' lits-of-l } (\text{trail } S'))$ 
    using card S S' undef by simp
    then have a1:  $\exists x. (\text{full cdcl}_W\text{-cp } S')$  using IH  $\langle \text{no-strange-atm } S' \rangle$  by blast
    have ?case
    proof -
    obtain S'' :: 'st where
    ff1:  $\text{cdcl}_W\text{-cp}^{**} S' S'' \wedge \text{no-step cdcl}_W\text{-cp } S''$ 
    using a1 unfolding full-def by blast
    have  $\text{cdcl}_W\text{-cp}^{**} S S''$ 
    using ff1  $\text{cdcl}_W\text{-cp.intros}(2)[\text{OF propagate}]$ 
    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  $\text{cdcl}_W\text{-cp.cases rtranclp.rtrancl-refl}$ )
qed

```

18.5.3 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**

$\text{no-clause-is-false} \equiv$

$\lambda S. (\text{conflicting } S = \text{None} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{\text{as}} \text{CNot } D))$

abbreviation $\text{conflict-is-false-with-level} :: 'st \Rightarrow \text{bool}$ **where**

$\text{conflict-is-false-with-level } S \equiv \forall D. \text{conflicting } S = \text{Some } D \longrightarrow D \neq \{\#\}$
 $\longrightarrow (\exists L \in \# D. \text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S)$

lemma $\text{not-conflict-not-any-negated-init-clss}$:

assumes $\forall S'. \neg \text{conflict } S S'$

shows $\text{no-clause-is-false } S$

proof (clarify)

fix D

assume $D \in \# \text{ local.clauses } S$ **and** $\text{raw-conflicting } S = \text{None}$ **and** $\text{trail } S \models_{\text{as}} \text{CNot } D$

moreover then obtain D' **where**

$\text{mset-cls } D' = D$ **and**

$D' \nmid \text{ raw-clauses } S$

using $\text{in-mset-clss-exists-preimage}$ **unfolding** raw-clauses-def **by** blast

ultimately show False

using $\text{conflict-rule}[\text{of } S D' \text{ update-conflicting } (\text{Some } (\text{ccls-of-cls } D')) S]$ **assms**

by auto

qed

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 *auto*

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 *auto*

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 elim: conflictE*)
next
 case *propagate'*
 then show *?case* by (*auto elim: propagateE*)

qed

lemma *no-chained-conflict*:

assumes *conflict S S'*
 and *conflict S' S''*
 shows *False*
 using *assms unfolding conflict.simps*
 by (*metis conflicting-update-conflicting option.distinct(1) option.simps(9) state-eq-conflicting*)

lemma *rtranclp-cdcl_W-cp-propa-or-propa-confl*:

assumes *cdcl_W-cp** S U*
 shows *propagate** S U \vee ($\exists T. propagate** S T \wedge conflict T U$)*
 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 elim: conflictE)
  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 rtranclp-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 = Some D and
    D: D  $\neq$  {#}
  consider (CT) conflicting S = None | (SD) D' where conflicting S = Some D'
  by (cases conflicting S) auto
  then show  $\exists L \in \#D. \text{get-level}(\text{trail } U) L = \text{backtrack-lvl } U$ 
  proof cases
    case SD
    then have S = U
      by (metis (no-types) assms(1) cdclW-cp-conflicting-not-empty full-def rtranclpD tranclpD)
    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 full unfolding full-def
      apply (metis (no-types) rtranclpD tranclp-cdclW-cp-no-more-init-clss)
      by (metis (mono-tags, lifting) full full-def rtranclp-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 rtranclpD)
      have  $\bigwedge p \text{ pa. } \neg \text{propagate } p \text{ pa} \vee \text{conflicting } p =$ 
        (None::'v clause option)
      by (auto elim: propagateE)
      then show ?thesis
        using f5 that tranclp-cdclW-cp-propagate-with-conflict-or-not[OF (cdclW-cp++ S U)]
        full confl CT unfolding full-def by auto
    qed
  obtain D' where
    raw-conflicting T = None and
    D': D'  $\notin$  raw-clauses T and
    tr: trail T  $\models$  as CNot (mset-cls D') and
    U: U  $\sim$  update-conflicting (Some (ccls-of-cls D')) T

```



```

    using TU by (auto elim!: conflictE)
have init-clss T = init-clss S and learned-clss T = learned-clss S
  using U (init-clss U = init-clss S) (learned-clss U = learned-clss S) by auto
then have D ∈# clauses S
  using confl U D' by (auto simp: raw-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) asms(1) full-def rtrancpl-cdclW-cp-drop While-trail)
  have trail U ⊨as CNot D
    using tr confl U by (auto elim!: conflictE)
ultimately obtain L where L ∈# D and ¬ L ∈ lits-of-l M
  unfolding tr-U CNot-def true-annot-def Ball-def true-annot-def true-cl-def by force

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 clause) marked-lit and
    xb :: ('v, nat, 'v clause) 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-l (trail S)
    using (¬ L ∈ lits-of-l M) (no-dup (trail U)) unfolding tr-U lits-of-def by auto
ultimately have get-level (trail U) L = 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 (rev M) 0 L = 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-l M
    using ⟨-L ∈ lits-of-l 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 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 tr-U
    by auto
  qed
then show ∃ L ∈ #D. get-level (trail U) L = backtrack-lvl U
  using ⟨L ∈ # D⟩ by blast
qed
qed

```

18.5.4 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 (trail S') L = 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 (trail S) L = 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 *lev-inv*: cdcl_W-M-level-inv S
shows no-more-propagation-to-do S'
using *assms*

proof –

obtain E L **where**

S: conflicting S = None **and**
 E: E !∈! raw-clauses S **and**
 LE: L ∈ # mset-cls E **and**
 tr: trail S ⊨_{as} CNot (mset-cls (remove-lit L E)) **and**
 undefL: undefined-lit (trail S) L **and**
 S': S' ∼ cons-trail (Propagated L E) S

using *propagate* **by** (elim propagateE) *simp*

let ?M' = Propagated L (mset-cls E) # trail S

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 (mset-cls E) # trail S)

using (trail S' = M2 @ M1) S' S undefL lev-inv

```

    by (cases M2) (auto simp:cdclW-M-level-inv-decomp)
  moreover {
    assume tl M2 @ M1 = trail S
    moreover have D + {#L'#} ∈# clauses S
      using D-L S S' undefL unfolding raw-clauses-def by auto
    moreover have get-maximum-possible-level M1 < backtrack-lvl S
      using get-max S S' undefL by auto
    ultimately obtain L' where L' ∈# D and
      get-level (trail S) L' = get-maximum-possible-level M1
      using H ⟨M1 ⊨as CNot D⟩ undef unfolding no-more-propagation-to-do-def by metis
    moreover
      { have cdclW-M-level-inv S'
        using cdclW-consistent-inv lev-inv cdclW.propagate[OF propagate] by blast
        then have no-dup ?M' using S' undefL unfolding cdclW-M-level-inv-def by auto
        moreover
          have atm-of L' ∈ atm-of ' (lits-of-l M1)
            using ⟨L' ∈# D⟩ ⟨M1 ⊨as CNot D⟩ by (metis atm-of-uminus image-eqI
              in-CNot-implies-uminus(2))
          then have atm-of L' ∈ atm-of ' (lits-of-l (trail S))
            using ⟨tl M2 @ M1 = trail S⟩[symmetric] S undefL by auto
          ultimately have atm-of L ≠ atm-of L' unfolding lits-of-def by auto
        }
      ultimately have ∃ L' ∈# D. get-level (trail S') L' = get-maximum-possible-level M1
        using S S' undefL by auto
    }
  moreover {
    assume M2 = [] and M1: M1 = Propagated L (mset-cls E) # trail S
    have cdclW-M-level-inv S'
      using cdclW-consistent-inv[OF lev-inv] cdclW.propagate[OF propagate] by blast
    then have get-all-levels-of-marked (trail S') = rev [Suc 0..<(Suc 0+backtrack-lvl S)]
      using S' undefL unfolding cdclW-M-level-inv-def by auto
    then have get-maximum-possible-level M1 = backtrack-lvl S'
      using get-maximum-possible-level-max-get-all-levels-of-marked[of M1] S' M1 undefL
      by (auto intro: Max-eqI)
    then have False using get-max by auto
  }
  ultimately show ∃ L. L ∈# D ∧ get-level (trail S') L = get-maximum-possible-level M1
    by fast
qed
qed

```

lemma *conflict-no-more-propagation-to-do:*

```

assumes
  conflict: conflict S S' and
  H: no-more-propagation-to-do S and
  M: cdclW-M-level-inv S
shows no-more-propagation-to-do S'
using assms unfolding no-more-propagation-to-do-def by (force elim!: conflictE)

```

lemma *cdcl_W-cp-no-more-propagation-to-do:*

```

assumes
  conflict: cdclW-cp S S' and
  H: no-more-propagation-to-do S and
  M: cdclW-M-level-inv S
shows no-more-propagation-to-do S'

```

```

using assms
proof (induct rule: cdclW-cp.induct)
case (conflict' S S')
then show ?case using conflict-no-more-propagation-to-do[of S S'] by blast
next
case (propagate' S S') note S = this
show 1: no-more-propagation-to-do S'
  using propagate-no-more-propagation-to-do[of S S'] S by blast
qed

lemma cdclW-then-exists-cdclW-stgy-step:
assumes
  o: cdclW-o S S' and
  alien: no-strange-atm S and
  lev: cdclW-M-level-inv S
shows  $\exists S'. \text{cdcl}_W\text{-stgy } S S'$ 
proof –
obtain S'' where full cdclW-cp S' S''
  using always-exists-full-cdclW-cp-step alien cdclW-no-strange-atm-inv cdclW-o-no-more-init-clss
  o other lev by (meson cdclW-consistent-inv)
then show ?thesis
  using assms by (metis always-exists-full-cdclW-cp-step cdclW-stgy.conflict' full-unfold other')
qed

lemma backtrack-no-decomp:
assumes
  S: raw-conflicting S = Some E and
  LE: L ∈ # mset-ccls E and
  L: get-level (trail S) L = backtrack-lvl S and
  D: get-maximum-level (trail S) (remove1-mset L (mset-ccls E)) < backtrack-lvl S and
  bt: backtrack-lvl S = get-maximum-level (trail S) (mset-ccls E) and
  M-L: cdclW-M-level-inv S
shows  $\exists S'. \text{cdcl}_W\text{-o } S S'$ 
proof –
have L-D: get-level (trail S) L = get-maximum-level (trail S) (mset-ccls E)
  using L D bt by (simp add: get-maximum-level-plus)
let ?i = get-maximum-level (trail S) (remove1-mset L (mset-ccls E))
obtain K M1 M2 where
  K: (Marked K (?i + 1) # M1, M2) ∈ set (get-all-marked-decomposition (trail S))
  using backtrack-ex-decomp[OF M-L, of ?i] D S by auto
show ?thesis using backtrack-rule[OF S LE K L] bt L bj cdclW-bj.simps by auto
qed

lemma cdclW-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: cdclW-learned-clause S and
  level-inv: cdclW-M-level-inv S and
  alien: no-strange-atm S and
  no-dup: distinct-cdclW-state S and
  conft: cdclW-conflicting S and
  conft-k: conflict-is-false-with-level S
shows (conflicting S = Some {#} ∧ unsatisfiable (set-mset (init-clss S)))
   $\vee$  (conflicting S = None ∧ trail S  $\models_{as}$  set-mset (init-clss S))

```

```

proof –
  let ?M = trail S
  let ?N = init-clss S
  let ?k = backtrack-lvl S
  let ?U = learned-clss S
  consider
    (None) raw-conflicting S = None
  | (Some-Empty) E where raw-conflicting S = Some E and mset-clls E = {#}
  | (Some) E' where raw-conflicting S = Some E' and
    conflicting S = Some (mset-clls E') and mset-clls E' ≠ {#}
  by (cases conflicting S, simp) auto
then show ?thesis
proof cases
  case (Some-Empty E)
  then have conflicting S = Some {#} by auto
  then have unsatisfiable (set-mset (init-clss S))
    using assms(3) unfolding cdclW-learned-clause-def true-clss-cll-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-cll-empty)
  then show ?thesis using Some-Empty by auto
next
  case None
  { assume ¬?M ⊨asm ?N
    have atm-of ‘ (lits-of-l ?M) = atms-of-mm ?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)
        moreover have conflicting S = None
          using None by auto
        ultimately have ∃ S'. cdclW-o S S'
          using cdclW-o.decide decide-rule ⟨l ∈ ?B⟩ no-strange-atm-def
          by (metis literal.sel(1) state-eq-def)
        then show False
          using termi cdclW-then-exists-cdclW-stgy-step[OF - alien] level-inv by blast
      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-l ?M)
    using ⟨D ∈# ?N⟩ unfolding ⟨atm-of ‘ (lits-of-l ?M) = atms-of-mm ?N⟩ atms-of-ms-def
    by (auto simp add: atms-of-def)
  then have a1: atm-of ‘ set-mset D ⊆ atm-of ‘ lits-of-l (trail S)
    by (auto simp add: atms-of-def lits-of-def)
  have total-over-m (lits-of-l ?M) {D}
    using ⟨atms-of D ⊆ atm-of ‘ (lits-of-l ?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-cll-true-clss-CNot ⟨¬ trail S ⊨a D⟩ true-annot-def
    true-annots-true-cll 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  $f2$   $\langle D \in \# \text{ init-clss } S \rangle$  None  $\langle \text{trail } S \models_{\text{as}} CNot\ D \rangle$ 
    raw-clauses-def full-cdclW-cp-not-any-negated-init-clss by auto
  qed
}
then have  $?M \models_{\text{asm}} ?N$  by blast
then show  $?thesis$ 
  using None by auto
next
case (Some  $E'$ ) note raw-conf = this(1) and LD = this(2) and nempty = this(3)
then obtain  $L$   $D$  where
   $E'[simp]$ : mset-ccls  $E' = D + \{\#L\#\}$  and
  lev-L: get-level  $?M$   $L = ?k$ 
  by (metis (mono-tags) confl-k insert-DiffM2)
let  $?D = D + \{\#L\#\}$ 
have  $?D \neq \{\#\}$  by auto
have  $?M \models_{\text{as}} CNot\ ?D$  using confl LD unfolding cdclW-conflicting-def by auto
then have  $?M \neq []$  unfolding true-annots-def Ball-def true-annot-def true-clss-def by force
have  $M$ :  $?M = hd\ ?M\ \# \ tl\ ?M$  using  $\langle ?M \neq [] \rangle$  list.collapse by fastforce
have g-a-l: get-all-levels-of-marked  $?M = rev\ [1..<1 + ?k]$ 
  using level-inv lev-L M unfolding cdclW-M-level-inv-def by auto
have g-k: get-maximum-level (trail  $S$ )  $D \leq ?k$ 
  using get-maximum-possible-level-ge-get-maximum-level[of ?M]
  get-maximum-possible-level-max-get-all-levels-of-marked[of ?M]
  by (auto simp add: Max-n-upt g-a-l)
{
  assume marked: is-marked (hd  $?M$ )
  then obtain  $k'$  where  $k': k' + 1 = ?k$ 
    using level-inv M unfolding cdclW-M-level-inv-def
    by (cases hd (trail S); cases trail S) auto
  obtain  $L'$   $l'$  where  $L': hd\ ?M = Marked\ L'\ l'$  using marked by (cases hd ?M) auto
  have marked-hd-tl: get-all-levels-of-marked (hd (trail  $S$ )  $\#$  tl (trail  $S$ ))
    = rev  $[1..<1 + \text{length}(\text{get-all-levels-of-marked } ?M)]$ 
    using level-inv lev-L M unfolding cdclW-M-level-inv-def M[symmetric]
    by blast
  then have  $l'-tl$ :  $l' \# \text{get-all-levels-of-marked } (tl\ ?M)$ 
    = 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
     $l'-cons$ :  $l' \# \text{get-all-levels-of-marked } (tl\ (\text{trail } S)) =$ 
       $\text{length}(\text{get-all-levels-of-marked } (\text{trail } S))$ 
       $\# \text{rev } [1..<\text{length}(\text{get-all-levels-of-marked } (\text{trail } S))]$  and
     $l' = ?k$  and
     $g-r$ : get-all-levels-of-marked (tl (trail  $S$ ))
      = rev  $[1..<\text{length}(\text{get-all-levels-of-marked } (\text{trail } S))]$ 
    using level-inv lev-L M unfolding cdclW-M-level-inv-def by auto

```

```

have *:  $\bigwedge list. no\_dup\ list \implies$ 
   $- L \in lits\_of\_l\ list \implies atm\_of\ L \in atm\_of\ ' lits\_of\_l\ list$ 
  by (metis atm-of-uminus imageI)
have L'-L:  $L' = -L$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  moreover have  $-L \in lits\_of\_l\ ?M$  using confl LD unfolding cdclW-conflicting-def by auto
  ultimately have get-level (hd (trail S) # tl (trail S))  $L = get\_level\ (tl\ ?M)\ L$ 
  using cdclW-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-l-cons marked-lit.sel(1))
  moreover
    have length (get-all-levels-of-marked (trail S)) = ?k
      using level-inv unfolding cdclW-M-level-inv-def by auto
    then have Max (set (0 # get-all-levels-of-marked (tl (trail S)))) = ?k - 1
      unfolding g-r by (auto simp add: Max-n-upt)
    then have get-level (tl ?M)  $L < ?k$ 
      using get-maximum-possible-level-ge-get-level[of tl ?M L]
      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 lev-L M by auto
qed
have L: hd ?M = Marked (-L) ?k using  $\langle l' = ?k \rangle L'-L\ L'$  by auto

have get-maximum-level (trail S)  $D < ?k$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then have get-maximum-level (trail S)  $D = ?k$  using M g-k unfolding L by auto
  then obtain L'' where  $L'' \in \# D$  and L-k: get-level ?M  $L'' = ?k$ 
    using get-maximum-level-exists-lit[of ?k ?M D] unfolding k'[symmetric] by auto
  have  $L \neq L''$  using no-dup  $\langle L'' \in \# D \rangle$ 
    unfolding distinct-cdclW-state-def LD
    by (metis E' add.right-neutral add-diff-cancel-right'
      distinct-mem-diff-mset union-commute union-single-eq-member)
  have  $L'' = -L$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have get-level ?M  $L'' = get\_level\ (tl\ ?M)\ L''$ 
      using M  $\langle L \neq L'' \rangle$  get-level-skip-beginning[of L'' hd ?M tl ?M] unfolding L
      by (auto simp: atm-of-eq-atm-of)
    then show False
      by (metis L-k Max-n-upt One-nat-def Suc-n-not-le-n  $\langle l' = backtrack\_lvl\ S \rangle$ 
        add-Suc-right add-implies-diff g-r
        get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked list.set(2)
        get-rev-level-less-max-get-all-levels-of-marked k' l'-cons list.sel(1)
        rev-rev-ident semiring-normalization-rules(6) set-upt)
  qed
  then have taut: tautology (D + {#L#})
    using  $\langle L'' \in \# D \rangle$  by (metis add commute mset-leD mset-le-add-left multi-member-this
      tautology-minus)
  have consistent-interp (lits-of-l ?M)
    using level-inv unfolding cdclW-M-level-inv-def by auto
  then have  $\neg ?M \models_{as} CNot\ ?D$ 
    using taut by (metis  $\langle L'' = -L \rangle \langle L'' \in \# D \rangle$  add commute consistent-interp-def)

```

```

    diff-union-cancelR in-CNot-implies-uminus(2) in-diffD multi-member-this)
  moreover have ?M  $\models_{as}$  CNot ?D
    using confl no-dup LD unfolding cdclW-conflicting-def by auto
  ultimately show False by blast
qed note H = this
have get-maximum-level (trail S) D < get-maximum-level (trail S) (D + {#L#})
  using H by (auto simp: get-maximum-level-plus lev-L max-def)
moreover have backtrack-lvl S = get-maximum-level (trail S) (D + {#L#})
  using H by (auto simp: get-maximum-level-plus lev-L max-def)
ultimately have False
  using backtrack-no-decomp[OF raw-conf - lev-L] level-inv termi
  cdclW-then-exists-cdclW-stgy-step[of S] alien unfolding E'
  by (auto simp add: lev-L max-def)
} note not-is-marked = this

moreover {
  let ?D = D + {#L#}
  have ?D  $\neq$  {#} by auto
  have ?M  $\models_{as}$  CNot ?D using confl LD unfolding cdclW-conflicting-def by auto
  then have ?M  $\neq$  [] unfolding true-annot-def Ball-def true-annot-def true-cls-def by force
  assume nm:  $\neg$ is-marked (hd ?M)
  then obtain L' C where L'C: hd-raw-trail S = Propagated L' C
    by (metis (trail S  $\neq$  []) hd-raw-trail is-marked-def mset-of-mlit.elims)
  then have hd ?M = Propagated L' (mset-cls C)
    using (trail S  $\neq$  []) hd-raw-trail mset-of-mlit.simps(1) by fastforce
  then have M: ?M = Propagated L' (mset-cls C) # tl ?M using (?M  $\neq$  []) list.collapse by
fastforce
  then obtain C' where C': mset-cls C = C' + {#L'#}
    using confl unfolding cdclW-conflicting-def by (metis append-Nil diff-single-eq-union)
  { assume  $-L' \notin \#$  ?D
    then have Ex (skip S)
      using skip-rule[OF M raw-conf] unfolding E' by auto
    then have False
      using cdclW-then-exists-cdclW-stgy-step[of S] alien level-inv termi
      by (auto dest: cdclW-o.intros cdclW-bj.intros)
  }
  moreover {
    assume L'D:  $-L' \in \#$  ?D
    then obtain D' where D': ?D = D' + {#-L'#} by (metis insert-DiffM2)
    have g-r: get-all-levels-of-marked (Propagated L' (mset-cls 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' (mset-cls 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 (trail S) D'  $\leq$  ?k
      using get-maximum-possible-level-ge-get-maximum-level[of
        Propagated L' (mset-cls C) # tl ?M] M
      unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
    then have get-maximum-level (trail S) D' = ?k
       $\vee$  get-maximum-level (trail S) D' < ?k
      using le-neq-implies-less by blast
    moreover {
      assume g-D'-k: get-maximum-level (trail S) D' = ?k

```



```

then have f1: get-maximum-level (trail S) D' = backtrack-lvl S
  using M by auto
then have Ex (cdclW-o S)
  using f1 resolve-rule[of S L' C , OF ⟨trail S ≠ []⟩ - - raw-conf] raw-conf g-D'-k
  L'C L'D unfolding C' D' E'
  by (fastforce simp add: D' intro: cdclW-o.intros cdclW-bj.intros)
then have False
  by (meson alien cdclW-then-exists-cdclW-stgy-step termi level-inv)
}
moreover {
  assume a1: get-maximum-level (trail S) D' < ?k
  then have f3: get-maximum-level (trail S) D' < get-level (trail S) (-L')
    using a1 lev-L by (metis D' get-maximum-level-ge-get-level insert-noteq-member
      not-less)
  moreover have backtrack-lvl S = get-level (trail S) L'
    apply (subst M)
    unfolding rev.simps
    apply (subst get-rev-level-can-skip-correctly-ordered)
    using level-inv unfolding cdclW-M-level-inv-def
    apply (subst (asm) (2) M) apply (simp add: cdclW-M-level-inv-decomp)
    using level-inv unfolding cdclW-M-level-inv-def
    apply (subst (asm) (2) M) apply (auto simp add: cdclW-M-level-inv-decomp)[]
    using level-inv unfolding cdclW-M-level-inv-def
    apply (subst (asm) (4) M) apply (auto simp add: cdclW-M-level-inv-decomp)[]
    using level-inv unfolding cdclW-M-level-inv-def
    apply (subst (asm) (4) M) by (auto simp add: cdclW-M-level-inv-decomp)[]
  moreover then have get-level (trail S) L' = get-maximum-level (trail S) (D' + {#- L'#})
    using a1 by (auto simp add: get-maximum-level-plus max-def)
  ultimately have False
    using M backtrack-no-decomp[of S - -L', OF raw-conf]
    cdclW-then-exists-cdclW-stgy-step L'D level-inv termi alien
    unfolding D' E' by auto
}
ultimately have False by blast
}
ultimately have False by blast
}
ultimately show ?thesis by blast
qed
qed

```

lemma *cdcl_W-cp-tranclp-cdcl_W:*
 $cdcl_W\text{-}cp\ S\ S' \implies cdcl_W^{++}\ S\ S'$
apply (*induct rule: cdcl_W-cp.induct*)
by (*meson cdcl_W.conflict cdcl_W.propagate tranclp.r-into-trancl tranclp.trancl-into-trancl*) $+$

lemma *tranclp-cdcl_W-cp-tranclp-cdcl_W:*
 $cdcl_W\text{-}cp^{++}\ S\ S' \implies cdcl_W^{++}\ S\ S'$
apply (*induct rule: tranclp.induct*)
apply (*simp add: cdcl_W-cp-tranclp-cdcl_W*)
by (*meson cdcl_W-cp-tranclp-cdcl_W tranclp-trans*)

lemma *cdcl_W-stgy-tranclp-cdcl_W:*
 $cdcl_W\text{-}stgy\ S\ S' \implies cdcl_W^{++}\ S\ S'$
proof (*induct rule: cdcl_W-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.other tranclp.r-into-trancl
    tranclp-cdclW-cp-tranclp-cdclW tranclp-trans)
qed

lemma tranclp-cdclW-stgy-tranclp-cdclW:
  cdclW-stgy++ S S'  $\implies$  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'  $\implies$  cdclW** S S'
  using rtranclp-unfold[of cdclW-stgy S S'] tranclp-cdclW-stgy-tranclp-cdclW[of S S'] by auto

lemma in-remove1-mset-neg:
  assumes ab: a ≠ b
  shows a ∈ # remove1-mset b C  $\longleftrightarrow$  a ∈ # C
proof -
  have count {#b#} a = 0
    using ab by simp
  then show ?thesis
    by (metis (no-types) count-diff diff-zero mem-Collect-eq set-mset-def)
qed

lemma not-empty-get-maximum-level-exists-lit:
  assumes n: D ≠ {#}
  and max: get-maximum-level M D = n
  shows ∃ L ∈ #D. get-level M L = n
proof -
  have f: finite (insert 0 ((λL. get-level M L) ' set-mset D)) by auto
  then have n ∈ ((λL. get-level M L) ' set-mset D)
    using n max get-maximum-level-exists-lit-of-max-level image-iff
    unfolding get-maximum-level-def by force
  then show ∃ L ∈ # D. get-level M L = n by auto
qed

lemma get-maximum-level-union-mset:
  get-maximum-level M (A #∪ B) = get-maximum-level M (A + B)
  unfolding get-maximum-level-def by (auto simp: image-Un)

lemma cdclW-o-conflict-is-false-with-level-inv:
  assumes
    cdclW-o S S' and
    lev: cdclW-M-level-inv S and
    confl-inv: conflict-is-false-with-level S and

```

n-d: *distinct-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(4)* **and** *LD = this(5)* **and** *T = this(7)*
 have *uL-not-D: -L ∉ # remove1-mset (-L) (mset-ccls D)*
 using *n-d confl unfolding distinct-cdcl_W-state-def distinct-mset-def*
 by (*metis distinct-cdcl_W-state-def distinct-mem-diff-mset multi-member-last n-d option.simps(9)*)
 moreover have *L-not-D: L ∉ # remove1-mset (-L) (mset-ccls D)*
 proof (*rule ccontr*)
 assume $\neg ?thesis$
 then have *L ∈ # mset-ccls D*
 by (*auto simp: in-remove1-mset-neg*)
 moreover have *Propagated L (mset-cls C) # M ⊨_{as} CNot (mset-ccls D)*
 using *conflicting confl tr-S unfolding cdcl_W-conflicting-def by auto*
 ultimately have *-L ∈ lits-of-l (Propagated L (mset-cls C) # M)*
 using *in-CNot-implies-uminus(2) by blast*
 moreover have *no-dup (Propagated L (mset-cls C) # 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) distinct-consistent-interp)*
 qed
ultimately
 have *g-D: get-maximum-level (Propagated L (mset-cls C) # M) (remove1-mset (-L) (mset-ccls D))*
 = get-maximum-level M (remove1-mset (-L) (mset-ccls D))
 proof –
 have $\forall a f L. ((a::'v) \in f \text{ ' } L) = (\exists l. (l::'v \text{ literal}) \in L \wedge a = f l)$
 by *blast*
 then show *?thesis*
 using *get-maximum-level-skip-first[of L remove1-mset (-L) (mset-ccls D) mset-cls C M]*
 unfolding *atms-of-def*
 by (*metis (no-types) uL-not-D L-not-D atm-of-eq-atm-of*)
 qed
 have *lev-L[simp]: get-level M L = 0*
 apply (*rule atm-of-notin-get-rev-level-eq-0*)
 using *lev unfolding cdcl_W-M-level-inv-def tr-S by auto*

 have *D: get-maximum-level M (remove1-mset (-L) (mset-ccls D)) = backtrack-lvl S*
 using *resolve.hyps(6) LD unfolding tr-S by (auto simp: get-maximum-level-plus max-def g-D)*
 have *get-all-levels-of-marked M = rev [Suc 0..*Suc (backtrack-lvl S)*]*
 using *lev unfolding tr-S cdcl_W-M-level-inv-def by auto*
 then have *get-maximum-level M (remove1-mset L (mset-cls C)) ≤ backtrack-lvl S*
 using *get-maximum-possible-level-ge-get-maximum-level[of M]*
 get-maximum-possible-level-max-get-all-levels-of-marked[of M] by (auto simp: Max-n-upt)
 then have *get-maximum-level M (remove1-mset (- L) (mset-ccls D) # ∪ remove1-mset L (mset-cls C)) =*
 backtrack-lvl S
 by (*auto simp: get-maximum-level-union-mset get-maximum-level-plus max-def D*)
 then show *?case*
 using *tr-S not-empty-get-maximum-level-exists-lit[of*
 remove1-mset (- L) (mset-ccls D) # ∪ remove1-mset L (mset-cls C) M] T
 by *auto*

```

next
case (skip L C' M D T) note tr-S = this(1) and D = this(2) and T = this(5)
then obtain La where La ∈# mset-ccls D and get-level (Propagated L C' # M) La = backtrack-lvl
S
  using skip confl-inv by auto
moreover
have atm-of La ≠ atm-of L
proof (rule ccontr)
  assume ¬ ?thesis
  then have La: La = L using ⟨La ∈# mset-ccls D⟩ ⟨¬ L ∈# mset-ccls D⟩
  by (auto simp add: atm-of-eq-atm-of)
  have Propagated L C' # M ⊨as CNot (mset-ccls D)
  using conflicting tr-S D unfolding cdclW-conflicting-def by auto
  then have −L ∈ lits-of-l M
  using ⟨La ∈# mset-ccls D⟩ in-CNot-implies-uminus(2)[of L mset-ccls D 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 (Propagated L C' # M) La = get-level M La by auto
ultimately show ?case using D tr-S T by auto
next
case backtrack
then show ?case
  by (auto split: if-split-asm simp: cdclW-M-level-inv-decomp lev)
qed auto

```

18.5.5 Strong completeness

```

lemma cdclW-cp-propagate-confl:
  assumes cdclW-cp S T
  shows propagate** S T ∨ (∃ S'. propagate** S S' ∧ conflict S' T)
  using assms by induction blast+

lemma rtrancp-cdclW-cp-propagate-confl:
  assumes cdclW-cp** S T
  shows propagate** S T ∨ (∃ S'. propagate** S S' ∧ conflict S' T)
  by (simp add: assms rtrancp-cdclW-cp-propa-or-propa-confl)

lemma propagate-high-levelE:
  assumes propagate S T
  obtains M' N' U k L C where
    state S = (M', N', U, k, None) and
    state T = (Propagated L (C + {#L#}) # M', N', U, k, None) and
    C + {#L#} ∈# local.clauses S and
    M' ⊨as CNot C and
    undefined-lit (trail S) L
proof −
  obtain E L where
    conf: conflicting S = None and
    E: E !∈! raw-clauses S and
    LE: L ∈# mset-cls E and
    tr: trail S ⊨as CNot (mset-cls (remove-lit L E)) and
    undef: undefined-lit (trail S) L and
    T: T ∼ cons-trail (Propagated L E) S
  using assms by (elim propagateE) simp

```

```

obtain  $M\ N\ U\ k$  where
   $S$ : state  $S = (M, N, U, k, \text{None})$ 
  using conf by auto
show thesis
  using that[of  $M\ N\ U\ k\ L\ \text{remove1-mset}\ L\ (\text{mset-cl}\ E)$ ]  $S\ T\ LE\ E\ tr\ \text{undef}$ 
  by auto
qed

lemma cdclW-cp-propagate-completeness:
  assumes  $MN$ :  $\text{set } M \models_s \text{set-mset } N$  and
  cons: consistent-interp ( $\text{set } M$ ) and
  tot: total-over-m ( $\text{set } M$ ) ( $\text{set-mset } N$ ) and
  lits-of-l ( $\text{trail } S$ )  $\subseteq \text{set } M$  and
  init-clss  $S = N$  and
  propagate**  $S\ S'$  and
  learned-clss  $S = \{\#\}$ 
  shows  $\text{length } (\text{trail } S) \leq \text{length } (\text{trail } S') \wedge \text{lits-of-l } (\text{trail } S') \subseteq \text{set } M$ 
  using assms(6,4,5,7)
proof (induction rule: rtranclp-induct)
  case base
  then show ?case by auto
next
  case (step  $Y\ Z$ )
  note  $st = \text{this}(1)$  and  $\text{propa} = \text{this}(2)$  and  $IH = \text{this}(3)$  and  $\text{lits}' = \text{this}(4)$  and  $NS = \text{this}(5)$  and
     $\text{learned} = \text{this}(6)$ 
  then have  $\text{len}: \text{length } (\text{trail } S) \leq \text{length } (\text{trail } Y)$  and  $LM: \text{lits-of-l } (\text{trail } Y) \subseteq \text{set } M$ 
    by blast+

obtain  $M'\ N'\ U\ k\ C\ L$  where
   $Y$ : state  $Y = (M', N', U, k, \text{None})$  and
   $Z$ : state  $Z = (\text{Propagated } L\ (C + \{\#L\}) \# M', N', U, k, \text{None})$  and
   $C: C + \{\#L\} \in \# \text{ clauses } Y$  and
   $M'-C: M' \models_{as} C\text{Not } C$  and
  undefined-lit ( $\text{trail } Y$ )  $L$ 
  using  $\text{propa}$  by (auto elim: propagate-high-levelE)
have init-clss  $S = \text{init-clss } Y$ 
  using  $st$  by induction (auto elim: propagateE)
then have [simp]:  $N' = N$  using  $NS\ Y\ Z$  by simp
have learned-clss  $Y = \{\#\}$ 
  using  $st$  learned by induction (auto elim: propagateE)
then have [simp]:  $U = \{\#\}$  using  $Y$  by auto
have  $\text{set } M \models_s C\text{Not } C$ 
  using  $M'-C\ LM\ Y$  unfolding true-annots-def Ball-def true-annot-def true-clss-def true-cl-def
  by force
moreover
  have  $\text{set } M \models C + \{\#L\}$ 
  using  $MN\ C\ \text{learned } Y\ NS\ \langle \text{init-clss } S = \text{init-clss } Y \rangle \langle \text{learned-clss } Y = \{\#\} \rangle$ 
  unfolding true-clss-def raw-clauses-def by fastforce
ultimately have  $L \in \text{set } M$  by (simp add: cons consistent-CNot-not)
then show ?case using  $LM\ \text{len}\ Y\ Z$  by auto
qed

lemma
  assumes propagate**  $S\ X$ 
  shows

```

rtrancp-propagate-init-clss: $\text{init-clss } X = \text{init-clss } S$ and
rtrancp-propagate-learned-clss: $\text{learned-clss } X = \text{learned-clss } S$
using *assms* **by** (*induction rule*: *rtrancp-induct*) (*auto elim*: *propagateE*)

lemma *completeness-is-a-full1-propagation*:

fixes $S :: 'st$ **and** $M :: 'v$ *literal list*
assumes MN : $\text{set } M \models_s \text{set-mset } N$
and *cons*: *consistent-interp* ($\text{set } M$)
and *tot*: *total-over-m* ($\text{set } M$) ($\text{set-mset } N$)
and *alien*: *no-strange-atm* S
and *learned*: $\text{learned-clss } S = \{\#\}$
and $\text{clsS}[\text{simp}]$: $\text{init-clss } S = N$
and *lits*: $\text{lits-of-l } (\text{trail } S) \subseteq \text{set } M$
shows $\exists S'. \text{propagate}^{**} S S' \wedge \text{full cdcl}_W\text{-cp } S S'$

proof –

obtain S' **where** *full*: $\text{full cdcl}_W\text{-cp } S S'$
using *always-exists-full-cdcl_W-cp-step alien* **by** *blast*
then consider (*propa*) $\text{propagate}^{**} S S'$
 $|$ (*confl*) $\exists X. \text{propagate}^{**} S X \wedge \text{conflict } X S'$
using *rtrancp-cdcl_W-cp-propagate-confl* **unfolding** *full-def* **by** *blast*
then show *?thesis*
proof cases
case *propa* **then show** *?thesis* **using** *full* **by** *blast*
next
case *confl*
then obtain X **where**
 X : $\text{propagate}^{**} S X$ **and**
 $X\text{conf}$: $\text{conflict } X S'$
by *blast*
have clsX : $\text{init-clss } X = \text{init-clss } S$
using X **by** (*blast dest*: *rtrancp-propagate-init-clss*)
have learnedX : $\text{learned-clss } X = \{\#\}$
using X **learned by** (*auto dest*: *rtrancp-propagate-learned-clss*)
obtain E **where**
 E : $E \in \# \text{init-clss } X + \text{learned-clss } X$ **and**
 $\text{Not-}E$: $\text{trail } X \models_{as} C\text{Not } E$
using $X\text{conf}$ **by** (*auto simp add*: *raw-clauses-def elim!*: *conflictE*)
have $\text{lits-of-l } (\text{trail } X) \subseteq \text{set } M$
using *cdcl_W-cp-propagate-completeness*[*OF assms(1–3) lits - X learned*] **learned by** *auto*
then have MNE : $\text{set } M \models_s C\text{Not } E$
using $\text{Not-}E$
by (*fastforce simp add*: *true-annots-def true-annot-def true-clss-def true-cl-def*)
have $\neg \text{set } M \models_s \text{set-mset } N$
using E *consistent-CNot-not*[*OF cons MNE*]
unfolding $\text{learnedX true-clss-def}$ **unfolding** clsX clsS **by** *auto*
then show *?thesis* **using** MN **by** *blast*
qed
qed

See also $\text{cdcl}_W\text{-cp}^{**} ?S ?S' \implies \exists M. \text{trail } ?S' = M @ \text{trail } ?S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$

lemma *rtrancp-propagate-is-trail-append*:

$\text{propagate}^{**} S T \implies \exists c. \text{trail } T = c @ \text{trail } S$
by (*induction rule*: *rtrancp-induct*) (*auto elim*: *propagateE*)

lemma *rtrancp-propagate-is-update-trail*:

$propagate^{**} S T \implies cdcl_W\text{-}M\text{-level-inv } S \implies$
 $init\text{-}class S = init\text{-}class T \wedge learned\text{-}class S = learned\text{-}class T \wedge backtrack\text{-}lvl S = backtrack\text{-}lvl T$
 $\wedge conflicting S = conflicting T$
proof (induction rule: *rtrancpl-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 *cdcl_W-M-level-inv U*
using *rtrancpl-cdcl_W-consistent-inv* $\langle propagate^{**} S T \rangle \langle propagate T U \rangle$
rtrancpl-mono[of *propagate cdcl_W*] *cdcl_W-cp-consistent-inv propagate'*
*rtrancpl-propagate-is-rtrancpl-cdcl_W step.prem*s **by** *blast*
then have *no-dup* (*trail U*) **unfolding** *cdcl_W-M-level-inv-def* **by** *auto*
ultimately show ?*case using* $\langle propagate T U \rangle$ **unfolding** *state-eq-def*
by (*fastforce simp: elim: propagateE*)
qed

lemma *cdcl_W-stgy-strong-completeness-n*:

assumes
MN: *set M* \models_s *set-mset* (*mset-class N*) **and**
cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset* (*mset-class N*)) **and**
atm-incl: *atm-of* ' (*set M*) \subseteq *atms-of-mm* (*mset-class N*) **and**
distM: *distinct M* **and**
length: $n \leq \text{length } M$
shows
 $\exists M' k S. \text{length } M' \geq n \wedge$
 $\text{lits-of-l } M' \subseteq \text{set } M \wedge$
 $\text{no-dup } M' \wedge$
 $\text{state } S = (M', \text{mset-class } N, \{\#\}, k, \text{None}) \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$
using *length*
proof (induction *n*)
case 0
have *state* (*init-state N*) = (\square , *mset-class N*, $\{\#\}$, 0, *None*)
by (*auto simp: state-eq-def simp del: state-simp*)
moreover have
 $0 \leq \text{length } \square$ **and**
 $\text{lits-of-l } \square \subseteq \text{set } M$ **and**
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) (\text{init-state } N)$
and *no-dup* \square
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**
 $\text{l-M': length } M' \geq n$ **and**
 $\text{M': lits-of-l } M' \subseteq \text{set } M$ **and**
 $\text{n-d[simp]: no-dup } M'$ **and**
 $\text{S: state } S = (M', \text{mset-class } N, \{\#\}, k, \text{None})$ **and**
 $\text{st: cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$
by *auto*
have
 $M: \text{cdcl}_W\text{-M-level-inv } S$ **and**
 $\text{alien: no-strange-atm } S$

```

    using cdclW-M-level-inv-S0-cdclW rtrancpl-cdclW-stgy-consistent-inv st apply blast
using cdclW-M-level-inv-S0-cdclW no-strange-atm-S0 rtrancpl-cdclW-no-strange-atm-inv
rtrancpl-cdclW-stgy-rtrancpl-cdclW st by blast

{ 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 simp: comp-def)
  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-l (trail S') ⊆ set M
    using S' full cdclW-cp-propagate-completeness[OF assms(1-3), of S] M' S
    by (auto simp: comp-def)
  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 simp: comp-def rev-map)
    with propa have length (trail S') > n
      using l-M' propa by (induction rule: trancpl.induct) (auto elim: propagateE)
  moreover
    have stS': cdclW-stgy** (init-state N) S'
      using st cdclW-stgy.conflict'[OF full] by auto
    then have init-clss S' = mset-clss 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' = None
      using trancpl-into-rtrancpl[OF ⟨propagate++ S S'⟩] S
      rtrancpl-propagate-is-update-trail[of S S'] S M unfolding state-eq-def
      by (auto simp: comp-def)
    have S-S': state S' = (trail S', mset-clss N, {#}, backtrack-lvl S', None)
      using S by auto
    have cdclW-stgy** (init-state N) S'
      apply (rule rtrancpl.rtrancpl-into-rtrancpl)
      using st apply simp
      using ⟨cdclW-stgy S S'⟩ 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' ≥ Suc n)
      case True

```



```

then show ?thesis using l-M' M' st M alien S n-d by blast
next
case False
then have n': length M' = n using l-M' by auto
have no-conf!: no-step conflict S
proof -
  { fix D
    assume D ∈# mset-cls N and M' ⊨as CNot D
    then have set M ⊨ D using MN unfolding true-cls-def by auto
    moreover have set M ⊨s CNot D
      using ⟨M' ⊨as CNot D⟩ M'
      by (metis le-iff-sup true-annots-true-cls true-cls-union-increase)
    ultimately have False using cons consistent-CNot-not by blast
  }
then show ?thesis
  using S by (auto simp: true-cls-def comp-def rev-map
    raw-clauses-def dest!: in-cls-mset-cls elim!: conflictE)
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 card (lits-of-l M') = length M'
  by (induction M') (auto simp add: lits-of-def card-insert-if)
then have lits-of-l M' ⊆ set M
  using n M' n' lenM by auto
then obtain m where m: m ∈ set M and undef-m: m ∉ lits-of-l M' by auto
moreover have undef: undefined-lit M' m
  using M' Marked-Propagated-in-iff-in-lits-of-l calculation(1,2) cons
    consistent-interp-def by (metis (no-types, lifting) subset-eq)
moreover have atm-of m ∈ atms-of-mm (init-cls S)
  using atm-incl calculation S by auto
ultimately
  have dec: decide S (cons-trail (Marked m (k+1)) (incr-lvl S))
    using decide-rule[of S -
      cons-trail (Marked m (k + 1)) (incr-lvl S)] S
    by auto
let ?S' = cons-trail (Marked m (k+1)) (incr-lvl S)
have lits-of-l (trail ?S') ⊆ set M using m M' S 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 rtranclp-mono[of decide cdclW] by (meson cdclW-consistent-inv decide other)
then have lev'': cdclW-M-level-inv S''
  using S'' rtranclp-cdclW-consistent-inv rtranclp-propagate-is-rtranclp-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-l (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-l (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'':
    state S'' = (trail S'', mset-cls N, {#}, backtrack-lvl S'', None)
    using rtrancp-propagate-is-update-trail[OF S''] S undef n-d'' lev''
    by auto
  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 (mset-cls N) and

cons: consistent-interp (set M) and

tot: total-over-m (set M) (set-mset (mset-cls N)) and

atm-incl: atm-of ' (set M) ⊆ atms-of-mm (mset-cls N) and

distM: distinct M

shows

∃ M' k S.

lits-of-l M' = set M ∧

state S = (M', mset-cls N, {#}, k, None) ∧

*cdcl_W-stgy** (init-state N) S ∧*

final-cdcl_W-state S

proof –

from *cdcl_W-stgy-strong-completeness-n*[OF *assms*, of length M]

obtain M' k T **where**

l: length M ≤ length M' and

M'-M: lits-of-l M' ⊆ set M and

no-dup: no-dup M' and

T: state T = (M', mset-cls N, {#}, k, None) and

*st: cdcl_W-stgy** (init-state N) T*

by *auto*

have *card (set M) = length M* **using** *distM* **by** (*simp add: distinct-card*)

moreover

have *cdcl_W-M-level-inv T*

using *rtrancp-cdcl_W-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-l 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-l M')* **using** *l unfolding lits-of-def* **by** *auto*

then have *set M = lits-of-l M'*

using *M'-M card-seteq* **by** *blast*

moreover

then have *M' ⊨_{asm} mset-cls N*

using *MN unfolding true-annots-def Ball-def true-annot-def true-cls-def* **by** *auto*

then have *final-cdcl_W-state T*

using *T no-dup unfolding final-cdcl_W-state-def* **by** *auto*

ultimately show ?thesis **using** *st T* **by** *blast*

qed

18.5.6 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-conf* ($S :: 'st$) \equiv
 $(\forall M K i M' D. M' @ \text{Marked } K i \# M = \text{trail } S \longrightarrow D \in \# \text{ clauses } S$
 $\longrightarrow \neg M \models_{as} CNot D)$

lemma *no-smaller-conf-init-sate*[simp]:
no-smaller-conf (init-state N) **unfolding** *no-smaller-conf-def* **by** *auto*

lemma *cdcl_W-o-no-smaller-conf-inv*:

fixes $S S' :: 'st$
assumes
cdcl_W-o $S S'$ **and**
lev: *cdcl_W-M-level-inv* S **and**
max-lev: *conflict-is-false-with-level* S **and**
smaller: *no-smaller-conf* S **and**
no-f: *no-clause-is-false* S
shows *no-smaller-conf* S'
using *assms*(1,2) **unfolding** *no-smaller-conf-def*
proof (*induct rule*: *cdcl_W-o-induct-lev2*)
case (*decide* $L T$) **note** *conf* = *this*(1) **and** *undef* = *this*(2) **and** $T = \text{this}(4)$
have [simp]: *clauses* $T = \text{clauses } S$
using T *undef* **by** *auto*
show ?*case*
proof (*intro allI impI*)
fix $M'' K i M' Da$
assume $M'' @ \text{Marked } K i \# M' = \text{trail } T$
and $Da \in \# \text{ local.clauses } T$
then have $tl M'' @ \text{Marked } K i \# M' = \text{trail } S$
 $\vee (M'' = [] \wedge \text{Marked } K i \# M' = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S)$
using T *undef* **by** (*cases* M'') *auto*
moreover {
assume $tl M'' @ \text{Marked } K i \# M' = \text{trail } S$
then have $\neg M' \models_{as} CNot Da$
using $D T$ *undef* *no-f* *conf* *smaller* **unfolding** *no-smaller-conf-def* *smaller* **by** *fastforce*
}
moreover {
assume $\text{Marked } K i \# M' = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S$
then have $\neg M' \models_{as} CNot Da$ **using** *no-f* D *conf* 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-conf-def* **by** *auto*
next
case *skip*
then show ?*case* **using** *smaller* *no-f* *max-lev* **unfolding** *no-smaller-conf-def* **by** *auto*
next
case (*backtrack* $K i M1 M2 L D T$) **note** *conf* = *this*(1) **and** $LD = \text{this}(2)$ **and** *decomp* = *this*(3)
and
 $\text{undef} = \text{this}(7)$ **and** $T = \text{this}(8)$
obtain c **where** $M: \text{trail } S = c @ M2 @ \text{Marked } K (i+1) \# M1$

```

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)
  let ?S' = (cons-trail (Propagated L (cls-of-ccls D))
    (reduce-trail-to M1 (add-learned-cls (cls-of-ccls D))
    (update-backtrack-lvl i (update-conflicting None S))))))
  assume D: Da ∈# clauses T
  moreover {
    assume Da ∈# clauses S
    then have  $\neg M \models_{as} CNot\ Da$  using  $\langle tl\ M' @\ Marked\ K'\ ia\ \# \ M = M1 \rangle$  M confl undef smaller
      unfolding no-smaller-confl-def by auto
  }
  moreover {
    assume Da: Da = mset-ccls D
    have  $\neg M \models_{as} CNot\ Da$ 
    proof (rule ccontr)
      assume  $\neg ?thesis$ 
      then have  $-L \in lits-of-l\ M$ 
        using LD unfolding Da by (simp add: in-CNot-implies-uminus(2))
      then have  $-L \in lits-of-l\ (Propagated\ L\ (mset-ccls\ D)\ \# M1)$ 
        using UnI2  $\langle tl\ M' @\ Marked\ K'\ ia\ \# \ M = M1 \rangle$ 
        by auto
      moreover
        have backtrack S ?S'
          using backtrack-rule[of S] backtrack.hyps
          by (force simp: state-eq-def simp del: state-simp)
        then have cdclW-M-level-inv ?S'
          using cdclW-consistent-inv[OF - lev] other[OF bj] by (auto intro: cdclW-bj.intros)
        then have no-dup (Propagated L (mset-ccls D)  $\# M1$ )
          using decomp undef lev unfolding cdclW-M-level-inv-def by auto
        ultimately show False by (metis consistent-interp-def distinct-consistent-interp
          insertCI lits-of-l-cons marked-lit.sel(2))
    qed
  }
  ultimately show  $\neg M \models_{as} CNot\ Da$ 
    using T undef decomp lev unfolding cdclW-M-level-inv-def by fastforce
qed
qed

lemma conflict-no-smaller-confl-inv:
  assumes conflict S S'
  and no-smaller-confl S
  shows no-smaller-confl S'
  using assms unfolding no-smaller-confl-def by (fastforce elim: conflictE)

lemma propagate-no-smaller-confl-inv:
  assumes propagate: propagate S S'
  and n-l: no-smaller-confl S
  shows no-smaller-confl S'

```

```

  unfolding no-smaller-conflict-def
proof (intro allI impI)
  fix M' K i M'' D
  assume M': M'' @ Marked K i # M' = trail S'
  and D ∈ # clauses S'
  obtain M N U k C L where
    S: state S = (M, N, U, k, None) and
    S': state S' = (Propagated L ( (C + {#L#}))) # M, N, U, k, None) and
    C + {#L#} ∈ # clauses S and
    M ⊨as CNot C and
    undefined-lit M L
  using propagate by (auto elim: propagate-high-levelE)
  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 ¬M' ⊨as CNot D
  using ⟨D ∈ # clauses S'⟩ n-l S S' raw-clauses-def unfolding no-smaller-conflict-def by auto
  then show ¬M' ⊨as CNot D by auto
qed

```

```

lemma 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: cdclW-cp.induct)
  case (conflict' S S')
  then show ?case using conflict-no-smaller-conflict-inv[of S S'] by blast
next
  case (propagate' S S')
  then show ?case using propagate-no-smaller-conflict-inv[of S S'] by fastforce
qed

```

```

lemma rtrancp-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: 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 (tranc1-into-tranc1 S S' S'')
 then show ?case using cdcl_W-cp-no-smaller-conf1-inv[of S' S''] by fast
 qed

lemma full-cdcl_W-cp-no-smaller-conf1-inv:
 assumes full cdcl_W-cp S S'
 and n-l: no-smaller-conf1 S
 shows no-smaller-conf1 S'
 using assms unfolding full-def
 using rtrancp-cdcl_W-cp-no-smaller-conf1-inv[of S S'] by blast

lemma full1-cdcl_W-cp-no-smaller-conf1-inv:
 assumes full1 cdcl_W-cp S S'
 and n-l: no-smaller-conf1 S
 shows no-smaller-conf1 S'
 using assms unfolding full1-def
 using trancp-cdcl_W-cp-no-smaller-conf1-inv[of S S'] by blast

lemma cdcl_W-stgy-no-smaller-conf1-inv:
 assumes cdcl_W-stgy S S'
 and n-l: no-smaller-conf1 S
 and conflict-is-false-with-level S
 and cdcl_W-M-level-inv S
 shows no-smaller-conf1 S'
 using assms

proof (induct rule: cdcl_W-stgy.induct)
 case (conflict' S')
 then show ?case using full1-cdcl_W-cp-no-smaller-conf1-inv[of S S'] by blast
 next
 case (other' S' S'')
 have no-smaller-conf1 S'
 using cdcl_W-o-no-smaller-conf1-inv[OF other'.hyps(1) other'.prems(3,2,1)]
 not-conflict-not-any-negated-init-clss other'.hyps(2) cdcl_W-cp.simps by auto
 then show ?case using full-cdcl_W-cp-no-smaller-conf1-inv[of S' S''] other'.hyps by blast
 qed

lemma is-conflicting-exists-conflict:
 assumes $\neg(\forall D \in \# \text{init-clss } S' + \text{learned-clss } S'. \neg \text{trail } S' \models_{\text{as}} \text{CNot } D)$
 and conflicting S' = None
 shows $\exists S''. \text{conflict } S' S''$
 using assms raw-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-conf1 S
 shows no-clause-is-false S'
 $\vee (\text{conflicting } S' = \text{None}$
 $\longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{backtrack-lvl } S'))$
 using assms(1,2)

```

proof (induct rule: cdclW-o-induct-lev2)
  case (decide L T) note  $S = \text{this}(1)$  and  $\text{undef} = \text{this}(2)$  and  $T = \text{this}(4)$ 
  show ?case
  proof (rule HOL.disjI2, clarify)
    fix D
    assume D:  $D \in \# \text{ clauses } T$  and M-D:  $\text{trail } T \models_{as} CNot \ D$ 
    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  $\neg L \in \# \ D$ 
    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  $L': x = \{\# - L'\# \}$   $L' \in \# \ D$  by auto
        obtain L'' where  $L'' \in \# \ x$  and  $\text{lits-of-l } (\text{Marked } L \ (\text{?k} + 1) \ \# \ ?M) \models_l L''$ 
          using M-D x T undef unfolding true-annots-def Ball-def true-annot-def CNot-def
            true-cls-def Bex-def by auto
        show  $\exists L \in \# \ x. \text{lits-of-l } ?M \models_l L$  unfolding Bex-def
          using L'(1) L'(2)  $\langle - L \notin \# \ D \rangle \langle L'' \in \# \ x \rangle$ 
             $\langle \text{lits-of-l } (\text{Marked } L \ (\text{backtrack-lvl } S + 1) \ \# \ \text{trail } S) \models_l L'' \rangle$  by auto
        qed
        then show False using  $\langle \neg ?M \models_{as} CNot \ D \rangle$  by auto
      qed
    have atm-of L  $\notin \text{atm-of } \langle \text{lits-of-l } ?M \rangle$ 
      using undef defined-lit-map unfolding lits-of-def by fastforce
    then have get-level (Marked L (?k + 1)  $\# \ ?M$ ) ( $\neg L$ ) = ?k + 1 by simp
    then show  $\exists La. La \in \# \ D \wedge \text{get-level } ?M' \ La = \text{backtrack-lvl } T$ 
      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(3) and undef = this(7) and  $T = \text{this}(8)$ 
  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:  $\text{trail } T = \text{Propagated } L \ (\text{mset-ccls } D) \ \# \ M1$ 
      using T decomp undef lev by (auto simp: cdclW-M-level-inv-decomp)
    have backtrack S T
      using backtrack-rule[of S] backtrack.hyps T

```

```

    by (force simp del: state-simp simp: state-eq-def)
  then have lev': cdclW-M-level-inv T
    using cdclW-consistent-inv lev other cdclW-bj.backtrack cdclW-o.bj by blast
  then have - L ∉ lits-of-l M1
    using lev cdclW-M-level-inv-def Marked-Propagated-in-iff-in-lits-of-l undef by blast
  { assume Da ∈# clauses S
    then have ¬M1 ⊨as CNot Da using no-l M unfolding no-smaller-conflict-def by auto
  }
  moreover {
    assume Da: Da = mset-ccls D
    have ¬M1 ⊨as CNot Da using ⟨- L ∉ lits-of-l M1⟩ unfolding Da
      using backtrack.hyps(2) in-CNot-implies-uminus(2) by auto
  }
  ultimately have ¬M1 ⊨as CNot Da
    using Da T undef decomp lev by (fastforce simp: cdclW-M-level-inv-decomp)
  then have -L ∈# Da
    using M-D ⟨- L ∉ lits-of-l M1⟩ in-CNot-implies-uminus(2)
      true-annots-CNot-lit-of-notin-skip T unfolding tr-T
    by (smt insert-iff lits-of-l-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 (mset-ccls D) # 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-l M1' unfolding lits-of-def by auto
  have get-level (Propagated L (mset-ccls D) # M1) (-L) = i
    using get-level-get-rev-level-get-all-levels-of-marked[OF L,
      of [Propagated L (mset-ccls D)]]
    by (simp add: g-M1 split: if-splits)
  then show ∃ La. La ∈# Da ∧ get-level (trail T) La = 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-conf: conflicting S = None and
  lev: cdclW-M-level-inv S
shows ∃ T. propagate** S T ∧ conflict T U
proof -
  consider (propa) propagate** S U
    | (confl) T where propagate** S T and conflict T U
  using full unfolding full-def by (blast dest: rtranclp-cdclW-cp-propa-or-propa-conf)
  then show ?thesis
  proof cases
    case confl
    then show ?thesis by blast
  next
    case propa
    then have conflicting U = None and
      [simp]: learned-clss U = learned-clss S and
      [simp]: init-clss U = init-clss S
      using no-conf rtranclp-propagate-is-update-trail lev by auto
    moreover

```



```

obtain  $D$  where  $D: D \in \# \text{clauses } U$  and
   $trS: \text{trail } S \models_{as} CNot\ D$ 
  using  $confl\ raw\text{-}clauses\text{-}def$  by  $auto$ 
obtain  $M$  where  $M: \text{trail } U = M @ \text{trail } S$ 
  using  $full\ rtrancp\ cdcl_W\text{-}cp\text{-}dropWhile\text{-}trail$  unfolding  $full\text{-}def$  by  $meson$ 
have  $tr\text{-}U: \text{trail } U \models_{as} CNot\ D$ 
  apply  $(rule\ true\text{-}annots\text{-}mono)$ 
  using  $trS$  unfolding  $M$  by  $simp\text{-}all$ 
have  $\exists V. \text{conflict } U\ V$ 
  using  $\langle \text{conflicting } U = None \rangle\ D\ raw\text{-}clauses\text{-}def\ not\text{-}conflict\text{-}not\text{-}any\text{-}negated\text{-}init\text{-}class\ tr\text{-}U$ 
  by  $meson$ 
then have  $False$  using  $full\ cdcl_W\text{-}cp.\text{conflict}'$  unfolding  $full\text{-}def$  by  $blast$ 
then show  $?thesis$  by  $fast$ 

```

qed
qed

lemma $full1\text{-}cdcl_W\text{-}cp\text{-}exists\text{-}conflict\text{-}full1\text{-}decompose$:

```

assumes
   $confl: \exists D \in \# \text{clauses } S. \text{trail } S \models_{as} CNot\ D$  and
   $full: full\ cdcl_W\text{-}cp\ S\ U$  and
   $no\text{-}confl: \text{conflicting } S = None$  and
   $lev: cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ 
shows  $\exists T\ D. \text{propagate}^{**}\ S\ T \wedge \text{conflict } T\ U$ 
   $\wedge \text{trail } T \models_{as} CNot\ D \wedge \text{conflicting } U = Some\ D \wedge D \in \# \text{clauses } S$ 

```

proof –

```

obtain  $T$  where  $propa: \text{propagate}^{**}\ S\ T$  and  $conf: \text{conflict } T\ U$ 
  using  $full1\text{-}cdcl_W\text{-}cp\text{-}exists\text{-}conflict\text{-}decompose[OF\ assms]$  by  $blast$ 
have  $p: \text{learned}\text{-}class\ T = \text{learned}\text{-}class\ S\ init\text{-}class\ T = init\text{-}class\ S$ 
  using  $propa\ lev\ rtrancp\text{-}propagate\text{-}is\text{-}update\text{-}trail$  by  $auto$ 
have  $c: \text{learned}\text{-}class\ U = \text{learned}\text{-}class\ T\ init\text{-}class\ U = init\text{-}class\ T$ 
  using  $conf$  by  $(auto\ elim: \text{conflict}E)$ 
obtain  $D$  where  $\text{trail } T \models_{as} CNot\ D \wedge \text{conflicting } U = Some\ D \wedge D \in \# \text{clauses } S$ 
  using  $conf\ p\ c$  by  $(fastforce\ simp: raw\text{-}clauses\text{-}def\ elim!: \text{conflict}E)$ 
then show  $?thesis$ 
  using  $propa\ conf$  by  $blast$ 

```

qed

lemma $cdcl_W\text{-}stgy\text{-}no\text{-}smaller\text{-}confl$:

```

assumes
   $cdcl_W\text{-}stgy\ S\ S'$  and
   $n\text{-}l: no\text{-}smaller\text{-}confl\ S$  and
   $conflict\text{-}is\text{-}false\text{-}with\text{-}level\ S$  and
   $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$  and
   $no\text{-}clause\text{-}is\text{-}false\ S$  and
   $distinct\text{-}cdcl_W\text{-}state\ S$  and
   $cdcl_W\text{-}conflicting\ S$ 

```

shows $no\text{-}smaller\text{-}confl\ S'$

using $assms$

proof $(induct\ rule: cdcl_W\text{-}stgy.induct)$

case $(\text{conflict}'\ S')$

show $no\text{-}smaller\text{-}confl\ S'$

using $\text{conflict}'.hyps\ \text{conflict}'.prems(1)\ full1\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv$ **by** $blast$

next

case $(other'\ S'\ S'')$

have $lev': cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$

```

    using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
show no-smaller-confl S''
    using cdclW-stgy-no-smaller-confl-inv[OF cdclW-stgy.other'[OF other'.hyps(1-3)]]
    other'.prems(1-3) by blast
qed

lemma cdclW-stgy-ex-lit-of-max-level:
  assumes
    cdclW-stgy S S' and
    n-l: no-smaller-confl S and
    conflict-is-false-with-level S and
    cdclW-M-level-inv S and
    no-clause-is-false S and
    distinct-cdclW-state S and
    cdclW-conflicting S
  shows conflict-is-false-with-level S'
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S')
  have no-smaller-confl S'
    using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-confl-inv by blast
  moreover have conflict-is-false-with-level S'
    using conflict'.hyps conflict'.prems(2-4)
    rtrancpl-cdclW-co-conflict-ex-lit-of-max-level[of S S']
    unfolding full-def full1-def rtrancpl-unfold by presburger
  then show ?case by blast
next
  case (other' S' S'')
  have lev': cdclW-M-level-inv S'
    using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
  moreover
    have no-clause-is-false S'
      ∨ (conflicting S' = None ⟶ (∀ D ∈ #clauses S'. trail S' ⊨as CNot D
        ⟶ (∃ L. L ∈ # D ∧ get-level (trail S') L = 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' = None
      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 rtrancpl-cdclW-co-conflict-ex-lit-of-max-level[of S' S''] lev' ⟨no-clause-is-false S'⟩
        by blast
    }
  }
  moreover
  {
    assume c: conflicting S' ≠ None
    have conflicting S ≠ None using other'.hyps(1) c
      by (induct rule: cdclW-o-induct) auto
    then have conflict-is-false-with-level S'
      using cdclW-o-conflict-is-false-with-level-inv[OF other'.hyps(1)]
      other'.prems(3,5,6,2) by blast
    moreover have cdclW-cp** S' S'' using other'.hyps(3) unfolding full-def by auto
  }

```

```

then have  $S' = S''$  using  $c$ 
  by (induct rule: rtranclp-induct)
    (fastforce intro: option.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  $S' = \text{None}$  and
     $D\text{-}L$ :  $\forall D \in \# \text{ clauses } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$ 
       $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{backtrack-lvl } S')$ 
  { assume  $\forall D \in \# \text{ clauses } S'. \neg \text{trail } S' \models_{\text{as}} \text{CNot } D$ 
    then have no-clause-is-false  $S'$  using confl by simp
    then have conflict-is-false-with-level  $S''$  using calculation(3) by presburger
  }
}
moreover {
  assume  $\neg(\forall D \in \# \text{ clauses } S'. \neg \text{trail } S' \models_{\text{as}} \text{CNot } D)$ 
  then obtain  $T D$  where
    propagate**  $S' T$  and
    conflict  $T S''$  and
     $D$ :  $D \in \# \text{ clauses } S'$  and
    trail  $S'' \models_{\text{as}} \text{CNot } D$  and
    conflicting  $S'' = \text{Some } D$ 
    using full1-cdclW-cp-exists-conflict-full1-decompose[OF - - confl]
    other'(3) lev' by (metis (mono-tags, lifting) conflictE state-eq-trail
      trail-update-conflicting)
  obtain  $M$  where  $M$ : trail  $S'' = M @ \text{trail } S'$  and  $nm$ :  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
    using rtranclp-cdclW-cp-dropWhile-trail other'(3) unfolding full-def by meson
  have btS: backtrack-lvl  $S'' = \text{backtrack-lvl } S'$ 
    using other'.hyps(3) unfolding full-def by (metis rtranclp-cdclW-cp-backtrack-lvl)
  have inv: cdclW-M-level-inv  $S''$ 
    by (metis (no-types) cdclW-stgy.conflict' cdclW-stgy-consistent-inv full-unfold lev'
      other'.hyps(3))
  then have nd: no-dup (trail  $S''$ )
    by (metis (no-types) cdclW-M-level-inv-decomp(2))
  have conflict-is-false-with-level  $S''$ 
  proof cases
    assume trail  $S' \models_{\text{as}} \text{CNot } D$ 
    moreover then obtain  $L$  where
       $L \in \# D$  and
      lev-L: get-level (trail  $S'$ )  $L = \text{backtrack-lvl } S'$ 
      using  $D\text{-}L$   $D$  by blast
    moreover
      have  $LS'$ :  $-L \in \text{lits-of-l } (\text{trail } S')$ 
        using  $\langle \text{trail } S' \models_{\text{as}} \text{CNot } D \rangle \langle L \in \# D \rangle$  in-CNot-implies-uminus(2) by blast
      { fix  $x :: ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit}$  and
         $xb :: ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit}$ 
        assume  $a1$ :  $x \in \text{set } (\text{trail } S')$  and
           $a2$ :  $xb \in \text{set } M$  and
           $a3$ :  $(\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ` set } M \cap (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ` set } (\text{trail } S') = \{\}$  and
           $a4$ :  $-L = \text{lit-of } x$  and
           $a5$ : atm-of  $L = \text{atm-of } (\text{lit-of } xb)$ 
        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  $\notin$  atm-of ' lits-of-l M
    using nd LS' unfolding M by (auto simp add: lits-of-def)
  then have get-level (trail S'') L = get-level (trail S') L
    unfolding M by (simp add: lits-of-def)
  ultimately show ?thesis using btS <conflicting S'' = Some D> by auto
next
assume  $\neg$ trail S'  $\models_{as}$  CNot D
then obtain L where L  $\in \#$  D and LM:  $-L \in$  lits-of-l M
  using <trail S''  $\models_{as}$  CNot D> unfolding M
    by (auto simp add: true-cls-def M true-annot-def true-annot-def
      split: if-split-asm)
{ fix x :: ('v, nat, 'v clause) marked-lit and
  xb :: ('v, nat, 'v clause) marked-lit
  assume a1: xb  $\in$  set (trail S') and
    a2: x  $\in$  set M and
    a3: atm-of L = atm-of (lit-of xb) and
    a4:  $-L =$  lit-of x and
    a5:  $(\lambda l. \text{atm-of (lit-of l)}) \text{ ' set } M \cap (\lambda l. \text{atm-of (lit-of l)}) \text{ ' set (trail S')}$ 
      = {}
  moreover have atm-of (lit-of xb) = atm-of ( $-L$ )
    using a3 by simp
  ultimately have False
    by auto }
then have LS': atm-of L  $\notin$  atm-of ' lits-of-l (trail S')
  using nd <L  $\in \#$  D> LM unfolding M by (auto simp add: lits-of-def)
show ?thesis
proof cases
  assume ne: get-all-levels-of-marked (trail S') = []
  have backtrack-lvl S'' = 0
    using inv ne nm unfolding cdclW-M-level-inv-def M
    by (simp add: get-all-levels-of-marked-nil-iff-not-is-marked)
  moreover
  have a1: get-level M L = 0
    using nm by auto
  then have get-level (M @ trail S') L = 0
    by (metis LS' get-all-levels-of-marked-nil-iff-not-is-marked
      get-level-skip-beginning-not-marked lits-of-def ne)
  ultimately show ?thesis using <conflicting S'' = Some D> <L  $\in \#$  D> unfolding M
    by auto
next
assume ne: get-all-levels-of-marked (trail S')  $\neq$  []
have hd (get-all-levels-of-marked (trail S')) = backtrack-lvl S'
  using ne lev' M nm unfolding cdclW-M-level-inv-def
  by (cases get-all-levels-of-marked (trail S'))
  (simp-all add: get-all-levels-of-marked-nil-iff-not-is-marked[symmetric])
moreover have atm-of L  $\in$  atm-of ' lits-of-l M
  using  $\neg L \in$  lits-of-l 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 <L  $\in \#$  D> <conflicting S'' = Some D>
    get-level-skip-beginning-hd-get-all-levels-of-marked[OF LS', of M]

```

```

      get-level-skip-in-all-not-marked[of rev M L backtrack-lvl S']
    unfolding lits-of-def btS M
  by auto
qed
qed
}
ultimately have conflict-is-false-with-level S'' by blast
}
moreover
{
  assume conflicting S' ≠ None
  have no-clause-is-false S' using ⟨conflicting S' ≠ None⟩ by auto
  then have conflict-is-false-with-level S'' using calculation(3) by presburger
}
ultimately show ?case by fast
qed

```

lemma *rtranclp-cdcl_W-stgy-no-smaller-confl-inv*:

```

  assumes
    cdclW-stgy** S S' and
    n-l: no-smaller-confl S and
    cls-false: conflict-is-false-with-level S and
    lev: cdclW-M-level-inv S and
    no-f: no-clause-is-false S and
    dist: distinct-cdclW-state S and
    conflicting: cdclW-conflicting S and
    decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
    learned: cdclW-learned-clause S and
    alien: no-strange-atm S
  shows no-smaller-confl S' ∧ conflict-is-false-with-level S'
  using assms(1)
proof (induct rule: rtranclp-induct)
  case base
  then show ?case using n-l cls-false by auto
next
  case (step S' S'') note st = this(1) and cdcl = this(2) and IH = this(3)
  have no-smaller-confl S' and conflict-is-false-with-level S'
  using IH by blast+
  moreover have cdclW-M-level-inv S'
  using st lev rtranclp-cdclW-stgy-rtranclp-cdclW
  by (blast intro: rtranclp-cdclW-consistent-inv)+
  moreover have no-clause-is-false S'
  using st no-f rtranclp-cdclW-stgy-not-non-negated-init-clss by presburger
  moreover have distinct-cdclW-state S'
  using rtanclp-distinct-cdclW-state-inv[of S S'] lev rtranclp-cdclW-stgy-rtranclp-cdclW[OF st]
  dist by auto
  moreover have cdclW-conflicting S'
  using rtranclp-cdclW-all-inv(6)[of S S'] st alien conflicting decomp dist learned lev
  rtranclp-cdclW-stgy-rtranclp-cdclW by blast
  ultimately show ?case
  using cdclW-stgy-no-smaller-confl[OF cdcl] cdclW-stgy-ex-lit-of-max-level[OF cdcl] by fast
qed

```

18.5.7 Final States are Conclusive

lemma *full-cdcl_W-stgy-final-state-conclusive-non-false*:

fixes $S' :: 'st$
assumes $full: full\ cdcl_W\text{-}stgy\ (init\text{-}state\ N)\ S'$
and $no\text{-}d: distinct\text{-}mset\text{-}mset\ (mset\text{-}class\ N)$
and $no\text{-}empty: \forall D \in \#mset\text{-}class\ N. D \neq \{\#\}$
shows $(conflicting\ S' = Some\ \{\#\} \wedge unsatisfiable\ (set\text{-}mset\ (init\text{-}class\ S')))$
 $\vee (conflicting\ S' = None \wedge trail\ S' \models_{asm}\ init\text{-}class\ S')$
proof –
let $?S = init\text{-}state\ N$
have
 $termi: \forall S''. \neg cdcl_W\text{-}stgy\ S'\ S''$ **and**
 $step: cdcl_W\text{-}stgy^{**}\ ?S\ S'$ **using** $full$ **unfolding** $full\text{-}def$ **by** $auto$
moreover have
 $learned: cdcl_W\text{-}learned\text{-}clause\ S'$ **and**
 $level\text{-}inv: cdcl_W\text{-}M\text{-}level\text{-}inv\ S'$ **and**
 $alien: no\text{-}strange\text{-}atm\ S'$ **and**
 $no\text{-}dup: distinct\text{-}cdcl_W\text{-}state\ S'$ **and**
 $confl: cdcl_W\text{-}conflicting\ S'$ **and**
 $decomp: all\text{-}decomposition\text{-}implies\text{-}m\ (init\text{-}class\ S')\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S'))$
using $no\text{-}d\ rtranclp\text{-}cdcl_W\text{-}stgy\text{-}rtranclp\text{-}cdcl_W[of\ ?S\ S']\ step\ rtranclp\text{-}cdcl_W\text{-}all\text{-}inv(1-6)[of\ ?S\ S']$
unfolding $rtranclp\text{-}unfold$ **by** $auto$
moreover
have $\forall D \in \#mset\text{-}class\ N. \neg [] \models_{as}\ CNot\ D$ **using** $no\text{-}empty$ **by** $auto$
then have $confl\text{-}k: conflict\text{-}is\text{-}false\text{-}with\text{-}level\ S'$
using $rtranclp\text{-}cdcl_W\text{-}stgy\text{-}no\text{-}smaller\text{-}confl\text{-}inv[OF\ step]\ no\text{-}d$ **by** $auto$
show $?thesis$
using $cdcl_W\text{-}stgy\text{-}final\text{-}state\text{-}conclusive[OF\ termi\ decomp\ learned\ level\text{-}inv\ alien\ no\text{-}dup\ confl\ confl\text{-}k]$.
qed

lemma $conflict\text{-}is\text{-}full1\text{-}cdcl_W\text{-}cp$:
assumes $cp: conflict\ S\ S'$
shows $full1\ cdcl_W\text{-}cp\ S\ S'$
proof –
have $cdcl_W\text{-}cp\ S\ S'$ **and** $conflicting\ S' \neq None$
using $cp\ cdcl_W\text{-}cp.intros$ **by** $(auto\ elim!: conflictE\ simp: state\text{-}eq\text{-}def\ simp\ del: state\text{-}simp)$
then have $cdcl_W\text{-}cp^{++}\ S\ S'$ **by** $blast$
moreover have $no\text{-}step\ cdcl_W\text{-}cp\ S'$
using $\langle conflicting\ S' \neq None \rangle$ **by** $(metis\ cdcl_W\text{-}cp\text{-}conflicting\text{-}not\text{-}empty\ option.exhaust)$
ultimately show $full1\ cdcl_W\text{-}cp\ S\ S'$ **unfolding** $full1\text{-}def$ **by** $blast+$
qed

lemma $cdcl_W\text{-}cp\text{-}fst\text{-}empty\text{-}conflicting\text{-}false$:
assumes
 $cdcl_W\text{-}cp\ S\ S'$ **and**
 $trail\ S = []$ **and**
 $conflicting\ S \neq None$
shows $False$
using $assms$ **by** $(induct\ rule: cdcl_W\text{-}cp.induct)\ (auto\ elim: propagateE\ conflictE)$

lemma $cdcl_W\text{-}o\text{-}fst\text{-}empty\text{-}conflicting\text{-}false$:
assumes $cdcl_W\text{-}o\ S\ S'$
and $trail\ S = []$
and $conflicting\ S \neq None$

```

shows False
using assms by (induct rule: cdclW-o-induct) auto

lemma cdclW-stgy-fst-empty-conflicting-false:
  assumes cdclW-stgy S S'
  and trail S = []
  and conflicting S ≠ None
  shows False
  using assms apply (induct rule: cdclW-stgy.induct)
  using tranclpD cdclW-cp-fst-empty-conflicting-false unfolding full1-def apply metis
  using cdclW-o-fst-empty-conflicting-false by blast
thm cdclW-cp.induct[split-format(complete)]

lemma cdclW-cp-conflicting-is-false:
  cdclW-cp S S' ⇒ conflicting S = Some {#} ⇒ False
  by (induction rule: cdclW-cp.induct) (auto elim: propagateE conflictE)

lemma rtranclp-cdclW-cp-conflicting-is-false:
  cdclW-cp++ S S' ⇒ conflicting S = Some {#} ⇒ False
  apply (induction rule: rtranclp.induct)
  by (auto dest: cdclW-cp-conflicting-is-false)

lemma cdclW-o-conflicting-is-false:
  cdclW-o S S' ⇒ conflicting S = Some {#} ⇒ False
  by (induction rule: cdclW-o-induct) auto

lemma cdclW-stgy-conflicting-is-false:
  cdclW-stgy S S' ⇒ conflicting S = Some {#} ⇒ False
  apply (induction rule: cdclW-stgy.induct)
  unfolding full1-def apply (metis (no-types) cdclW-cp-conflicting-not-empty tranclpD)
  unfolding full-def by (metis conflict-with-false-implies-terminated other)

lemma rtranclp-cdclW-stgy-conflicting-is-false:
  cdclW-stgy* S S' ⇒ conflicting S = Some {#} ⇒ S' = S
  apply (induction rule: rtranclp.induct)
  apply simp
  using cdclW-stgy-conflicting-is-false by blast

lemma full-cdclW-init-clss-with-false-normal-form:
  assumes
    ∀ m ∈ set M. ¬is-marked m and
    E = Some D and
    state S = (M, N, U, 0, E)
  full cdclW-stgy S S' and
  all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))
  cdclW-learned-clause S
  cdclW-M-level-inv S
  no-strange-atm S
  distinct-cdclW-state S
  cdclW-conflicting S
  shows ∃ M''. state S' = (M'', N, U, 0, Some {#})
  using assms(10,9,8,7,6,5,4,3,2,1)
proof (induction M arbitrary: E D S)
  case Nil
  then show ?case

```

```

    using rtrancp-cdclW-stgy-conflicting-is-false unfolding full-def cdclW-conflicting-def
    by fastforce
next
case (Cons L M) note IH = this(1) and full = this(8) and E = this(10) and inv = this(2-7) and
  S = this(9) and nm = this(11)
obtain K p where K: L = Propagated K p
  using nm by (cases L) auto
have every-mark-is-a-conflict S using inv unfolding cdclW-conflicting-def by auto
then have MpK: M  $\models$ 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: multiset-eq-iff)
then have K': L = Propagated K ((p - {#K#}) + {#K#})
  using K by auto
obtain p' where
  p': hd-raw-trail S = Propagated K p' and
  pp': mset-cls p' = p
  using hd-raw-trail[of S] S K by (cases hd-raw-trail S) auto
obtain raw-D where
  raw-D: raw-conflicting S = Some raw-D
  using S E by (cases raw-conflicting S) auto
then have raw-DD: mset-ccls raw-D = D
  using S E by auto
consider (D) D = {#} | (D') D  $\neq$  {#} by blast
then show ?case
  proof cases
    case D
    then show ?thesis
      using full rtrancp-cdclW-stgy-conflicting-is-false S unfolding full-def E D by auto
  next
    case D'
    then have no-p: no-step propagate S and no-c: no-step conflict S
      using S E by (auto elim: propagateE conflictE)
    then have no-step cdclW-cp S by (auto simp: cdclW-cp.simps)
    have res-skip:  $\exists T. (resolve S T \wedge no\text{-}step\ skip S \wedge full\ cdcl_W\text{-}cp\ T\ T) \vee (skip S T \wedge no\text{-}step\ resolve S \wedge full\ cdcl_W\text{-}cp\ T\ T)$ 
    proof cases
      assume  $\neg lit\text{-}of\ L \notin \# D$ 
      then obtain T where sk: skip S T
        using S D' K skip-rule unfolding E by fastforce
      then have res: no-step resolve S
        using  $\langle \neg lit\text{-}of\ L \notin \# D \rangle$  S D' K hd-raw-trail[of S] unfolding E
        by (auto elim!: skipE resolveE)
      have full cdclW-cp T T
        using sk by (auto intro!: option-full-cdclW-cp elim: skipE)
      then show ?thesis
        using sk res by blast
    next
      assume LD:  $\neg \neg lit\text{-}of\ L \notin \# D$ 
      then have D: Some D = Some ((D - {#-lit-of L#}) + {#-lit-of L#})
        by (auto simp add: multiset-eq-iff)

      have  $\bigwedge L. get\text{-}level\ M\ L = 0$ 
        by (simp add: nm)
      then have get-maximum-level (Propagated K (p - {#K#} + {#K#}) # M) (D - {#-

```



```

K#}) = 0
  using LD get-maximum-level-exists-lit-of-max-level
  proof -
    obtain L' where get-level (L#M) L' = get-maximum-level (L#M) D
      using LD get-maximum-level-exists-lit-of-max-level[of D L#M] by fastforce
    then show ?thesis by (metis (mono-tags) K' get-level-skip-all-not-marked
      get-maximum-level-exists-lit nm not-gr0)
    qed
  then obtain T where sk: resolve S T
    using resolve-rule[of S K p' raw-D] S p' ⟨K ∈# p⟩ raw-D LD
    unfolding K' D E pp' raw-DD by auto
  then have res: no-step skip S
    using LD S D' K hd-row-trail[of S] unfolding E
    by (auto elim!: skipE resolveE)
  have full cdclW-cp T T
    using sk by (auto simp: option-full-cdclW-cp elim: resolveE)
  then show ?thesis
    using sk res by blast
  qed
then have step-s: ∃ T. cdclW-stgy S T
  using ⟨no-step cdclW-cp S⟩ other' by (meson bj resolve skip)
have get-all-marked-decomposition (L # M) = [([], L#M)]
  using nm unfolding K apply (induction M rule: marked-lit-list-induct, simp)
  by (rename-tac L l xs, case-tac hd (get-all-marked-decomposition xs), auto)+
then have no-b: no-step backtrack S
  using nm S by (auto elim: backtrackE)
have no-d: no-step decide S
  using S E by (auto elim: decideE)

have full-S-S: full cdclW-cp S S
  using S E by (auto simp add: option-full-cdclW-cp)
then have no-f: no-step (full1 cdclW-cp) S
  unfolding full-def full1-def rtrancp-unfold by (meson trancpD)
obtain T where
  s: cdclW-stgy S T and st: cdclW-stgy** T S'
  using full step-s full unfolding full-def by (metis rtrancp-unfold trancpD)
have resolve S T ∨ skip S T
  using s no-b no-d res-skip full-S-S cdclW-cp-state-eq-compatible resolve-unique
  skip-unique unfolding cdclW-stgy.simps cdclW-o.simps full-unfold
  full1-def by (blast dest!: trancpD elim!: cdclW-bj.cases)+
then obtain D' where T: state T = (M, N, U, 0, Some D')
  using S E by (auto elim!: skipE resolveE simp: state-eq-def simp del: state-simp)

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 (mset-clss N)
  and empty: {#} ∈ # (mset-clss N)
  shows conflicting S' = Some {#} ∧ unsatisfiable (set-mset (init-clss 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 rtranclp-unfold by auto
  have ∃ S''. conflict ?S S''
    using empty not-conflict-not-any-negated-init-clss[of init-state N] by auto

  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-rtranclpE)
  have st: cdclW** ?S St
    by (simp add: rtranclp-unfold ⟨cdclW-stgy ?S St⟩ cdclW-stgy-tranclp-cdclW)

  have ∃ T. conflict ?S T
    using empty not-conflict-not-any-negated-init-clss[of ?S] 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 rtranclp-cdclW-cp-backtrack-lvl unfolding full1-def
    by (fastforce dest!: tranclp-into-rtranclp)
  have cls-St: init-clss St = mset-clss N
    using fullSt cdclW-stgy-no-more-init-clss[OF St] by auto
  have conflicting St ≠ None
  proof (rule ccontr)
    assume conf: ¬ ?thesis
    obtain E where
      ES: E !∈! raw-init-clss St and
      E: mset-cls E = {#}
    using empty cls-St by (metis in-mset-clss-exists-preimage)
    then have ∃ T. conflict St T
      using empty cls-St conflict-rule[of St E] ES conf unfolding E
      by (auto simp: raw-clauses-def dest: in-mset-clss-exists-preimage)
    then show False using fullSt unfolding full1-def by blast
  qed

  have 1: ∀ m ∈ set (trail St). ¬ is-marked m
    using fullSt unfolding full1-def by (auto dest!: tranclp-into-rtranclp
      rtranclp-cdclW-cp-dropWhile-trail)

```

```

have 2: full cdclW-stgy St S'
  using ⟨cdclW-stgy** St S'⟩ ⟨no-step cdclW-stgy S'⟩ bt unfolding full-def by auto
have 3: all-decomposition-implies-m
  (init-clss St)
  (get-all-marked-decomposition
   (trail St))
  using rtrancpl-cdclW-all-inv(1)[OF st] no-d bt by simp
have 4: cdclW-learned-clause St
  using rtrancpl-cdclW-all-inv(2)[OF st] no-d bt bt by simp
have 5: cdclW-M-level-inv St
  using rtrancpl-cdclW-all-inv(3)[OF st] no-d bt by simp
have 6: no-strange-atm St
  using rtrancpl-cdclW-all-inv(4)[OF st] no-d bt by simp
have 7: distinct-cdclW-state St
  using rtrancpl-cdclW-all-inv(5)[OF st] no-d bt by simp
have 8: cdclW-conflicting St
  using rtrancpl-cdclW-all-inv(6)[OF st] no-d bt by simp
have init-clss S' = init-clss St and conflicting S' = Some {#}
  using ⟨conflicting St ≠ None⟩ full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St]
  2 3 4 5 6 7 8 St apply (metis ⟨cdclW-stgy** St S'⟩ rtrancpl-cdclW-stgy-no-more-init-clss)
  using ⟨conflicting St ≠ None⟩ full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St - -
  S'] 2 3 4 5 6 7 8 by (metis bt option.exhaust prod.inject)

moreover have init-clss S' = mset-clss N
  using ⟨cdclW-stgy** (init-state N) S'⟩ rtrancpl-cdclW-stgy-no-more-init-clss by fastforce
moreover have unsatisfiable (set-mset (mset-clss N))
  by (meson empty satisfiable-def true-clss-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 (mset-clss N)
shows (conflicting S' = Some {#} ∧ unsatisfiable (set-mset (init-clss S')))
  ∨ (conflicting S' = None ∧ trail S' ⊨asm init-clss S')
using assms full-cdclW-stgy-final-state-conclusive-is-one-false
full-cdclW-stgy-final-state-conclusive-non-false by blast

```

lemma full-cdcl_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 (mset-clss N)
shows (conflicting S' = Some {#} ∧ unsatisfiable (set-mset (mset-clss N)))
  ∨ (conflicting S' = None ∧ trail S' ⊨asm (mset-clss N) ∧ satisfiable (set-mset (mset-clss N)))
proof -
  have N: init-clss S' = (mset-clss N)
    using full unfolding full-def by (auto dest: rtrancpl-cdclW-stgy-no-more-init-clss)
  consider
    (confl) conflicting S' = Some {#} and unsatisfiable (set-mset (init-clss S'))
  | (sat) conflicting S' = None and trail S' ⊨asm init-clss S'
  using full-cdclW-stgy-final-state-conclusive[OF assms] by auto
then show ?thesis
  proof cases
    case confl

```

```

    then show ?thesis by (auto simp: N)
  next
    case sat
    have  $cdcl_W$ -M-level-inv (init-state N) by auto
    then have  $cdcl_W$ -M-level-inv S'
      using full rtracp-cdclW-stgy-consistent-inv unfolding full-def by blast
    then have consistent-interp (lits-of-l (trail S')) unfolding  $cdcl_W$ -M-level-inv-def by blast
    moreover have lits-of-l (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
end
end
theory CDCL-W-Termination
imports CDCL-W
begin

context conflict-driven-clause-learningW
begin

```

18.6 Termination

The condition that no learned clause is a tautology is overkill (in the sense that the no-duplicate condition is enough), but we can reuse *simple-clss*.

The invariant contains all the structural invariants that holds,

definition $cdcl_W$ -all-struct-inv where

```

 $cdcl_W$ -all-struct-inv S  $\longleftrightarrow$ 
  no-strange-atm S  $\wedge$ 
   $cdcl_W$ -M-level-inv S  $\wedge$ 
  ( $\forall s \in \#$  learned-clss S.  $\neg$ tautology s)  $\wedge$ 
  distinct-cdclW-state S  $\wedge$ 
   $cdcl_W$ -conflicting S  $\wedge$ 
  all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))  $\wedge$ 
   $cdcl_W$ -learned-clause S

```

lemma $cdcl_W$ -all-struct-inv-inv:

assumes $cdcl_W$ S S' and $cdcl_W$ -all-struct-inv S

shows $cdcl_W$ -all-struct-inv S'

unfolding $cdcl_W$ -all-struct-inv-def

proof (intro HOL.conjI)

show no-strange-atm S'

using $cdcl_W$ -all-inv[OF assms(1)] assms(2) unfolding $cdcl_W$ -all-struct-inv-def by auto

show $cdcl_W$ -M-level-inv S'

using $cdcl_W$ -all-inv[OF assms(1)] assms(2) unfolding $cdcl_W$ -all-struct-inv-def by fast

show distinct-cdcl_W-state S'

using $cdcl_W$ -all-inv[OF assms(1)] assms(2) unfolding $cdcl_W$ -all-struct-inv-def by fast

show $cdcl_W$ -conflicting S'

using $cdcl_W$ -all-inv[OF assms(1)] assms(2) unfolding $cdcl_W$ -all-struct-inv-def by fast

show all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))

using $cdcl_W$ -all-inv[OF assms(1)] assms(2) unfolding $cdcl_W$ -all-struct-inv-def by fast

show $cdcl_W$ -learned-clause S'

using $cdcl_W$ -all-inv[OF assms(1)] assms(2) unfolding $cdcl_W$ -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 *cdcl_W-all-struct-inv-def* **by** *fast*
qed

lemma *rtrancpl-cdcl_W-all-struct-inv-inv*:
assumes *cdcl_W** S S'* **and** *cdcl_W-all-struct-inv S*
shows *cdcl_W-all-struct-inv S'*
using *assms* **by** *induction (auto intro: cdcl_W-all-struct-inv-inv)*

lemma *cdcl_W-stgy-cdcl_W-all-struct-inv*:
cdcl_W-stgy S T \implies cdcl_W-all-struct-inv S \implies cdcl_W-all-struct-inv T
by (*meson cdcl_W-stgy-trancpl-cdcl_W rttrancpl-cdcl_W-all-struct-inv-inv rttrancpl-unfold*)

lemma *rtrancpl-cdcl_W-stgy-cdcl_W-all-struct-inv*:
*cdcl_W-stgy** S T \implies cdcl_W-all-struct-inv S \implies cdcl_W-all-struct-inv T*
by (*induction rule: rttrancpl-induct*) (*auto intro: cdcl_W-stgy-cdcl_W-all-struct-inv*)

18.7 No Relearning of a clause

lemma *cdcl_W-o-new-clause-learned-is-backtrack-step*:
assumes *learned: D \in # learned-clss T* **and**
new: D \notin # learned-clss S **and**
cdcl_W: cdcl_W-o S T **and**
lev: cdcl_W-M-level-inv S
shows *backtrack S T \wedge conflicting S = Some D*
using *cdcl_W lev learned new*
proof (*induction rule: cdcl_W-o-induct-lev2*)
case (*backtrack K i M1 M2 L C T*) **note** *decomp = this(3)* **and** *undef = this(6)* **and** *andef = this(7)*
and
T = this(8) **and** *D-T = this(9)* **and** *D-S = this(10)*
then have *D = mset-ccls C*
using *not-gr0 lev* **by** (*auto simp: cdcl_W-M-level-inv-decomp*)
then show *?case*
using *T backtrack.hyps(1-5) backtrack.intros[OF backtrack.hyps(1,2)] backtrack.hyps(3-6)*
by *auto*
qed *auto*

lemma *cdcl_W-cp-new-clause-learned-has-backtrack-step*:
assumes *learned: D \in # learned-clss T* **and**
new: D \notin # learned-clss S **and**
cdcl_W: cdcl_W-stgy S T **and**
lev: cdcl_W-M-level-inv S
shows $\exists S'. \text{backtrack } S S' \wedge \text{cdcl}_W\text{-stgy}^{**} S' T \wedge \text{conflicting } S = \text{Some } D$
using *cdcl_W learned new*
proof (*induction rule: cdcl_W-stgy.induct*)
case (*conflict' S'*)
then show *?case*
unfolding *full1-def* **by** (*metis (mono-tags, lifting) rttrancpl-cdcl_W-cp-learned-clause-inv*
trancpl-into-rttrancpl)
next
case (*other' S' S''*)
then have *D \in # learned-clss S'*
unfolding *full-def* **by** (*auto dest: rttrancpl-cdcl_W-cp-learned-clause-inv*)
then show *?case*
using *cdcl_W-o-new-clause-learned-is-backtrack-step[OF - $\langle D \notin \# \text{learned-clss } S \rangle \langle \text{cdcl}_W\text{-o } S S' \rangle]$*

$\langle \text{full } \text{cdcl}_W\text{-cp } S' S'' \rangle \text{ lev by } (\text{metis } \text{cdcl}_W\text{-stgy.conflict' full-unfold r-into-rtrancpl}$
 $\text{rtrancpl.rtrancpl-refl})$
qed

lemma *rtrancpl-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 $\exists S' S''. \text{cdcl}_W\text{-stgy}^{**} S S' \wedge \text{backtrack } S' S'' \wedge \text{conflicting } S' = \text{Some } D \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} S'' T$
using *cdcl_W learned new*
proof (*induction rule: rtrancpl-induct*)
case *base*
then show *?case by blast*
next
case (*step T U*) **note** *st = this(1) and o = this(2) and IH = this(3) and*
D-U = this(4) and D-S = this(5)
show *?case*
proof (*cases D ∈ # learned-clss T*)
case *True*
then obtain *S' S'' where*
*st': cdcl_W-stgy** S S' and*
bt: backtrack S' S'' and
confl: conflicting S' = Some D and
*st'': cdcl_W-stgy** S'' T*
using *IH D-S by metis*
have *cdcl_W-stgy⁺⁺ S'' U*
using *st'' o by force*
then show *?thesis*
by (*meson bt confl rtrancpl-unfold st'*)
next
case *False*
have *cdcl_W-M-level-inv T*
using *lev rtrancpl-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 = Some 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 = Some 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 *Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')*
using *assms by (auto elim: propagateE)*

```

lemma conflict-no-more-Marked-lit:
  assumes conflict S S'
  shows Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')
  using assms by (auto elim: conflictE)

lemma cdclW-cp-no-more-Marked-lit:
  assumes cdclW-cp S S'
  shows Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')
  using assms apply (induct rule: cdclW-cp.induct)
  using conflict-no-more-Marked-lit propagate-no-more-Marked-lit by auto

lemma rtrancpl-cdclW-cp-no-more-Marked-lit:
  assumes cdclW-cp** S S'
  shows Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')
  using assms apply (induct rule: rtrancpl-induct)
  using cdclW-cp-no-more-Marked-lit by blast+

lemma cdclW-o-no-more-Marked-lit:
  assumes cdclW-o S S' and lev: cdclW-M-level-inv S and  $\neg \text{decide } S S'$ 
  shows Marked K i ∈ set (trail S') ⟶ Marked K i ∈ set (trail S)
  using assms
proof (induct rule: cdclW-o-induct-lev2)
  case backtrack note decomp = this(3) and undef = this(7) and T = this(8)
  then show ?case using lev by (auto simp: cdclW-M-level-inv-decomp)
next
  case (decide L T)
  then show ?case using decide-rule[OF decide.hyps] by blast
qed auto

lemma cdclW-new-marked-at-beginning-is-decide:
  assumes cdclW-stgy S S' and
  lev: cdclW-M-level-inv S and
  trail S' = M' @ Marked L i # M and
  trail S = M
  shows  $\exists T. \text{decide } S T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$ 
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S') note st = this(1) and no-dup = this(2) and S' = this(3) and S = this(4)
  have cdclW-M-level-inv S'
  using full1-cdclW-cp-consistent-inv no-dup st by blast
  then have Marked L i ∈ set (trail S') and Marked L i ∉ set (trail S)
  using no-dup unfolding S S' cdclW-M-level-inv-def by (auto simp add: rev-image-eqI)
  then have False
  using st rtrancpl-cdclW-cp-no-more-Marked-lit[of S S']
  unfolding full1-def rtrancpl-unfold by blast
  then show ?case by fast
next
  case (other' T U) note o = this(1) and ns = this(2) and st = this(3) and no-dup = this(4) and
  S' = this(5) and S = this(6)
  have cdclW-M-level-inv U
  by (metis (full-types) lev cdclW.simps cdclW-consistent-inv full-def o
  other'.hyps(3) rtrancpl-cdclW-cp-consistent-inv)
  then have Marked L i ∈ set (trail U) and Marked L i ∉ set (trail S)
  using no-dup unfolding S S' cdclW-M-level-inv-def by (auto simp add: rev-image-eqI)
  then have Marked L i ∈ set (trail T)

```

```

    using st rtrancp-cdclW-cp-no-more-Marked-lit unfolding full-def by blast
  then show ?case
    using cdclW-o-no-more-Marked-lit[OF o] ⟨Marked L i ∉ set (trail S)⟩ ns lev by meson
qed

```

lemma *cdcl_W-o-is-decide*:

```

  assumes cdclW-o S T and lev: cdclW-M-level-inv S
  trail T = drop (length M0) M' @ Marked L i # H @ M and
  ¬ (∃ M'. trail S = M' @ Marked L i # H @ M)
  shows decide S T
  using assms
proof (induction rule:cdclW-o-induct-lev2)
  case (backtrack K i M1 M2 L D T)
  then obtain c where trail S = c @ M2 @ Marked K (Suc i) # M1
  by auto
  show ?case
  using backtrack lev
  apply (cases drop (length M0) M')
  apply (auto simp: cdclW-M-level-inv-decomp)
  using ⟨trail S = c @ M2 @ Marked K (Suc i) # M1⟩
  by (auto simp: cdclW-M-level-inv-decomp)
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 cdclW-stgy** R U and
  trail U = M' @ Marked L i # H @ M and
  trail R = M and
  cdclW-M-level-inv R
  shows
    ∃ S T T'. cdclW-stgy** R S ∧ decide S T ∧ cdclW-stgy** T U ∧ cdclW-stgy** S U ∧
    no-step cdclW-cp S ∧ trail T = Marked L i # H @ M ∧ trail S = H @ M ∧ cdclW-stgy S T' ∧
    cdclW-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 = this(1) and IH = this(3) and s = this(2) and
    U = this(4) and S = this(5) and lev = this(6)
  show ?case
  proof (cases ∃ M'. trail T = M' @ Marked L i # H @ M)
    case False
    with s show ?thesis using U s st S
  proof induction
    case (conflict' W) note cp = this(1) and nd = this(2) and W = this(3)
    then obtain M0 where trail W = M0 @ trail T and nmarked: ∀ l ∈ set M0. ¬ is-marked l
    using rtrancp-cdclW-cp-dropWhile-trail unfolding full1-def rtrancp-unfold by meson
    then have MV: M' @ Marked L i # H @ M = M0 @ trail T unfolding W by simp
    then have V: trail T = drop (length M0) (M' @ Marked L i # H @ M)
    by auto
    have takeWhile (Not o is-marked) M' = M0 @ takeWhile (Not o is-marked) (trail T)
    using arg-cong[OF MV, of takeWhile (Not o is-marked)] nmarked

```



```

    by (simp add: takeWhile-tail)
  from arg-cong[OF this, of length] have length  $M_0 \leq \text{length } M'$ 
    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  $M_0$  where trail  $U = M_0 @ \text{trail } T'$  and nmarked:  $\forall l \in \text{set } M_0. \neg \text{is-marked } l$ 
    using rtranclp-cdclW-cp-dropWhile-trail cp unfolding full-def by meson
  then have MV:  $M' @ \text{Marked } L \ i \ \# \ H @ M = M_0 @ \text{trail } T'$  unfolding U by simp
  then have V: trail  $T' = \text{drop } (\text{length } M_0) (M' @ \text{Marked } L \ i \ \# \ H @ M)$ 
    by auto
  have takeWhile (Not o 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 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 tr-T': trail  $T' = \text{drop } (\text{length } M_0) M' @ \text{Marked } L \ i \ \# \ H @ M$  using V by auto
  then have LT': Marked  $L \ i \in \text{set } (\text{trail } T')$  by auto
  moreover
    have cdclW-M-level-inv T
      using lev rtranclp-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-rtranclp rtranclp.rtrancl-refl)+
  have [simp]: drop (length  $M_0$ )  $M' = []$ 
    using <decide T T'> <Marked  $L \ i \in \text{set } (\text{trail } T')$ > nd tr-T'
    by (auto simp add: Cons-eq-append-conv elim: decideE)
  have T': drop (length  $M_0$ )  $M' @ \text{Marked } L \ i \ \# \ H @ M = \text{Marked } L \ i \ \# \ \text{trail } T$ 
    using <decide T T'> <Marked  $L \ i \in \text{set } (\text{trail } T')$ > nd tr-T'
    by (auto elim: decideE)
  have trail T' = Marked  $L \ i \ \# \ \text{trail } T$ 
    using <decide T T'> <Marked  $L \ i \in \text{set } (\text{trail } T')$ > tr-T'
    by (auto elim: decideE)
  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  $L \ i \ \# \ \text{trail } T$ >
      <trail T' = drop (length  $M_0$ )  $M' @ \text{Marked } L \ i \ \# \ H @ M$ > 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:  $cdcl_W\text{-stgy}^{**} R S'$  and
2: decide  $S' S''$  and
3:  $cdcl_W\text{-stgy}^{**} S'' T$  and
4: no-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 *rtranclp-cdcl_W-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 rtranclp-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 rtranclp-trans[OF  $st$ ] rtranclp-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'* *rtranclp-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*(3) and *undef* = *this*(7) and
T = *this*(8) and *trT* = *this*(12)
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 *rtrancpl-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: rtrancpl-induct*) auto

lemma *remove1-mset-eq-remove1-mset-same:*

$\text{remove1-mset } L \ D = \text{remove1-mset } L' \ D \implies L \in \# \ D \implies L = L'$
by (*metis diff-single-trivial insert-DiffM multi-drop-mem-not-eq single-eq-single union-right-cancel*)

lemma *cdcl_W-o-cannot-learn:*

assumes
cdcl_W-o $y \ z$ **and**
lev: *cdcl_W-M-level-inv* y **and**
trM: $\text{trail } y = c @ \text{Marked } Kh \ i \ \# \ H$ **and**
DL: $D \notin \# \text{learned-clss } y$ **and**
LD: $L \in \# \ D$ **and**
DH: $\text{atms-of } (\text{remove1-mset } L \ D) \subseteq \text{atm-of 'lits-of-l } H$ **and**
LH: $\text{atm-of } L \notin \text{atm-of 'lits-of-l } H$ **and**
learned: $\forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{\text{as}} CNot \ T$ **and**
 $z: \text{trail } z = c' @ \text{Marked } Kh \ i \ \# \ H$

shows $D \notin \# \text{learned-clss } z$
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** *confl* = *this*(1) **and** *LD'* = *this*(2) **and** *decomp* = *this*(3)
and *levL* = *this*(4) **and** *levD* = *this*(5) **and** $j = \text{this}(6)$ **and** *undef* = *this*(7) **and** $T = \text{this}(8)$ **and**
 $z = \text{this}(14)$

obtain $M3$ **where** $M3: \text{trail } y = M3 @ M2 @ \text{Marked } K \ (\text{Suc } j) \ \# \ M1$

using *decomp* *get-all-marked-decomposition-exists-prepend* **by** *metis*

have $M: \text{trail } y = c @ \text{Marked } Kh \ i \ \# \ H$ **using** *trM* **by** *simp*

have $H: \text{get-all-levels-of-marked } (\text{trail } y) = \text{rev } [1..<1 + \text{backtrack-lvl } y]$

using *lev* *unfolding cdcl_W-M-level-inv-def* **by** *auto*

have $c' @ \text{Marked } Kh \ i \ \# \ H = \text{Propagated } L' \ (\text{mset-ccls } D') \ \# \ \text{trail } (\text{reduce-trail-to } M1 \ y)$

using z *decomp undef T lev* **by** (*force simp: cdcl_W-M-level-inv-def*)

then obtain d **where** $d: M1 = d @ \text{Marked } Kh \ i \ \# \ H$

by (*metis (no-types) decomp in-get-all-marked-decomposition-trail-update-trail list.inject list.sel(3) marked-lit.distinct(1) self-append-conv2 tl-append2*)

have $i \in \text{set } (\text{get-all-levels-of-marked } (M3 @ M2 @ \text{Marked } K \ (\text{Suc } j) \ \# \ d @ \text{Marked } Kh \ i \ \# \ H))$
by *auto*

then have $i > 0$ **unfolding** $H[\text{unfolded } M3 \ d]$ **by** *auto*

show ?case

proof

assume $D \in \# \text{learned-clss } T$

then have $DLD': D = \text{mset-ccls } D'$

using $DL \ T \text{neq0-conv undef decomp lev}$ **by** (*fastforce simp: cdcl_W-M-level-inv-def*)

have $L\text{-cKh}: \text{atm-of } L \in \text{atm-of 'lits-of-l } (c @ [\text{Marked } Kh \ i])$

using $LH \text{learned } M \ DLD'[\text{symmetric}] \text{confl } LD' \ LD$

```

apply (auto simp add: image-iff dest!: in-CNot-implies-uminus)
apply (metis atm-of-uminus)+ done
have get-all-levels-of-marked (M3 @ M2 @ Marked K (j + 1) # M1)
  = rev [1.. $1 + \text{backtrack-lvl } y$ ]
using lev unfolding cdclW-M-level-inv-def M3 by auto
from arg-cong[OF this, of  $\lambda a. (\text{Suc } j) \in \text{set } a$ ] have backtrack-lvl  $y \geq j$  by auto

have DD'[simp]: remove1-mset L D = mset-ccls D' - {#L'#}
proof (rule ccontr)
  assume DD':  $\neg ?thesis$ 
  then have L'  $\in \#$  remove1-mset L D using DLD' LD by (metis LD' in-remove1-mset-neq)
  then have get-level (trail y) L'  $\leq$  get-maximum-level (trail y) (remove1-mset L D)
    using get-maximum-level-ge-get-level by blast
  moreover {
    have get-maximum-level (trail y) (remove1-mset L D) =
      get-maximum-level H (remove1-mset L D)
      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]  $\langle i > 0 \rangle$  by auto
      ultimately have get-maximum-level (trail y) (remove1-mset L D)  $< i$ 
        by (metis (full-types) dual-order.strict-trans nat-neq-iff not-le
          get-maximum-possible-level-ge-get-maximum-level) }
    moreover
      have L  $\in \#$  remove1-mset L' (mset-ccls D')
        using DLD'[symmetric] DD' LD by (metis in-remove1-mset-neq)
      then have get-maximum-level (trail y) (remove1-mset L' (mset-ccls D'))  $\geq$ 
        get-level (trail y) L
        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 (trail y) L = get-level (c @ [Marked Kh i]) L
        using L-cKh LH unfolding M by simp
      have get-level (c @ [Marked Kh i]) L  $\geq i$ 
        using L-cKh levL
         $\langle \text{get-all-levels-of-marked } (c @ [\text{Marked } Kh \ i]) = \text{rev } [i..< \text{backtrack-lvl } y + 1] \rangle$ 
        calculation(1,2) by auto
      then have get-level (trail y) L  $\geq i$ 
        using M  $\langle \text{get-level } (trail \ y) \ L = \text{get-level } (c @ [\text{Marked } Kh \ i]) \ L \rangle$  by auto }
    moreover
      have get-maximum-level (trail y) (remove1-mset L' (mset-ccls D'))
         $< \text{get-level } (trail \ y) \ L$ 
        using  $\langle j \leq \text{backtrack-lvl } y \rangle \text{ levL } j \text{ calculation}(1-4)$  by linarith
      ultimately show False using backtrack.hyps(4) by linarith

```

qed
then have $LL': L = L'$
using $LD\ LD'\ remove1-mset-eq-remove1-mset-same$ unfolding $DLD'[symmetric]$ by fast
have $nd: no-dup\ (trail\ y)$ using lev unfolding $cdcl_W-M-level-inv-def$ by auto

{ assume $D: remove1-mset\ L\ (mset-ccls\ D') = \{\#\}$
then have $j: j = 0$ using $levD\ j$ by (simp add: LL')
have $\forall m \in set\ M1. \neg is-marked\ m$
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 $D[simp]: remove1-mset\ L\ (mset-ccls\ D') \neq \{\#\}$
have $i \leq j$
using H unfolding $M3\ d$ by (auto simp add: $rev-swap[symmetric]$
dest: $upt-decomp-lt$)
have $j > 0$ apply (rule $ccontr$)
using $H\ \langle i > 0 \rangle$ unfolding $M3\ d$
by (auto simp add: $rev-swap[symmetric]$ dest!: $upt-decomp-lt$)
obtain L'' where
 $L'' \in \# remove1-mset\ L\ (mset-ccls\ D')$ and
 $L''D': get-level\ (trail\ y)\ L'' = get-maximum-level\ (trail\ y)$
 $(remove1-mset\ L\ (mset-ccls\ D'))$
using $get-maximum-level-exists-lit-of-max-level[OF\ D,\ of\ trail\ y]$ by auto
have $L''M: atm-of\ L'' \in atm-of\ 'lits-of-l\ (trail\ y)$
using $get-rev-level-ge-0-atm-of-in[of\ 0\ rev\ (trail\ y)\ L'']\ \langle j > 0 \rangle\ levD\ L''D'$
 $\langle j \leq backtrack-lvl\ y \rangle\ levL$ by (simp add: $LL'\ j$)
then have $L'' \in lits-of-l\ (Marked\ Kh\ i\ \# d)$
proof -
{
assume $L''H: atm-of\ L'' \in atm-of\ 'lits-of-l\ H$
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 $get-level\ (trail\ y)\ L'' = get-level\ H\ L''$
using $L''H$ unfolding M by simp
ultimately have $False$
using $levD\ \langle j > 0 \rangle\ get-rev-level-in-levels-of-marked[of\ rev\ H\ 0\ L'']\ \langle i \leq j \rangle$
unfolding $L''D'[symmetric]\ nd$
by (metis $L''D'\ LL'\ Max-n-upt\ Nat.le-trans\ One-nat-def\ Suc-pred\ \langle 0 < i \rangle$
 $get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked$
 $get-rev-level-less-max-get-all-levels-of-marked\ j\ lessI\ list.simps(15)$
 $not-less\ rev-rev-ident\ set-upt$)
}
moreover
have $atm-of\ L'' \in atm-of\ 'lits-of-l\ H$
using $DD'\ DH\ \langle L'' \in \# remove1-mset\ L\ (mset-ccls\ D') \rangle\ atm-of-lit-in-atms-of\ LL'\ LD$
 LD' by fastforce
ultimately show ?thesis
using $DD'\ DH\ \langle L'' \in \# remove1-mset\ L\ (mset-ccls\ D') \rangle\ atm-of-lit-in-atms-of$
by auto
}

qed
moreover

have $\text{atm-of } L'' \in \text{atms-of } (\text{remove1-mset } L (\text{mset-ccls } D'))$
using $\langle L'' \in \# \text{ remove1-mset } L (\text{mset-ccls } D') \rangle$ **by** $(\text{auto simp: atms-of-def})$

then have $\text{atm-of } L'' \in \text{atm-of ' lits-of-l } H$
using DH **unfolding** DD' **unfolding** LL' **by** blast
ultimately have False
using nd **unfolding** $M3$ d LL' **by** auto
}
ultimately show False **by** blast
qed
qed auto

lemma $\text{cdcl}_W\text{-stgy-with-trail-end-has-not-been-learned}$:

assumes
 $\text{cdcl}_W\text{-stgy } y \ z$ **and**
 $\text{cdcl}_W\text{-M-level-inv } y$ **and**
 $\text{trail } y = c @ \text{Marked } Kh \ i \ \# \ H$ **and**
 $D \notin \# \text{ learned-clss } y$ **and**
 $LD: L \in \# \ D$ **and**
 $DH: \text{atms-of } (\text{remove1-mset } L \ D) \subseteq \text{atm-of ' lits-of-l } H$ **and**
 $LH: \text{atm-of } L \notin \text{atm-of ' lits-of-l } H$ **and**
 $\forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{\text{as}} CNot \ T$ **and**
 $\text{trail } z = c' @ \text{Marked } Kh \ i \ \# \ H$
shows $D \notin \# \text{ learned-clss } z$
using assms
proof induction
case conflict'
then show $?case$
unfolding full1-def **using** $\text{trancpl-cdcl}_W\text{-cp-learned-clause-inv}$ **by** auto
next
case $(\text{other' } T \ U)$ **note** $o = \text{this}(1)$ **and** $cp = \text{this}(3)$ **and** $lev = \text{this}(4)$ **and** $\text{trY} = \text{this}(5)$ **and**
 $\text{notin} = \text{this}(6)$ **and** $LD = \text{this}(7)$ **and** $DH = \text{this}(8)$ **and** $LH = \text{this}(9)$ **and** $\text{confl} = \text{this}(10)$ **and**
 $\text{trU} = \text{this}(11)$
obtain c' **where** $c': \text{trail } T = c' @ \text{Marked } Kh \ i \ \# \ H$
using cp $\text{beginning-not-marked-invert}[of \ - \ \text{trail } T \ c' \ Kh \ i \ H]$
 $\text{rtrancpl-cdcl}_W\text{-cp-dropWhile-trail}[of \ T \ U]$ **unfolding** trU full-def **by** fastforce
show $?case$
using $\text{cdcl}_W\text{-o-cannot-learn}[OF \ o \ lev \ \text{trY} \ \text{notin} \ LD \ DH \ LH \ \text{confl} \ c']$
 $\text{rtrancpl-cdcl}_W\text{-cp-learned-clause-inv } cp$ **unfolding** full-def **by** auto
qed

lemma $\text{rtrancpl-cdcl}_W\text{-stgy-with-trail-end-has-not-been-learned}$:

assumes
 $(\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } K \ i \ \# \ H @ []))^{**} S \ z$ **and**
 $\text{cdcl}_W\text{-all-struct-inv } S$ **and**
 $\text{trail } S = c @ \text{Marked } K \ i \ \# \ H$ **and**
 $D \notin \# \text{ learned-clss } S$ **and**
 $LD: L \in \# \ D$ **and**
 $DH: \text{atms-of } (\text{remove1-mset } L \ D) \subseteq \text{atm-of ' lits-of-l } H$ **and**
 $LH: \text{atm-of } L \notin \text{atm-of ' lits-of-l } H$ **and**
 $\exists c'. \text{trail } z = c' @ \text{Marked } K \ i \ \# \ H$
shows $D \notin \# \text{ learned-clss } z$
using $\text{assms}(1-4, 8)$
proof $(\text{induction rule: rtrancpl-induct})$
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 cdclW** S T
proof -
  have  $\forall p \text{ pa. } \exists s \text{ sa. } \forall sb \text{ sc } sd \text{ se. } (\neg p^{**} (sb::'st) \text{ sc} \vee p \text{ s sa} \vee pa^{**} sb \text{ sc})$ 
     $\wedge (\neg pa \text{ s sa} \vee \neg p^{**} sd \text{ se} \vee pa^{**} sd \text{ se})$ 
  by (metis (no-types) mono-rtrancpl)
  then have cdclW-stgy** S T
  using st by blast
  then show ?thesis
  using rtrancpl-cdclW-stgy-rtrancpl-cdclW by blast
qed
then have lev': cdclW-all-struct-inv T
  using rtrancpl-cdclW-all-struct-inv-inv[of S T] lev by auto
then have confl':  $\forall Ta. \text{conflicting } T = \text{Some } Ta \longrightarrow \text{trail } T \models_{as} CNot \text{ } Ta$ 
  unfolding cdclW-all-struct-inv-def cdclW-conflicting-def by blast
show ?case
  apply (rule cdclW-stgy-with-trail-end-has-not-been-learned[OF - - c - LD DH LH confl' c])
  using s lev' IH c unfolding cdclW-all-struct-inv-def by blast+
qed

lemma cdclW-stgy-new-learned-clause:
  assumes cdclW-stgy S T and
    lev: cdclW-M-level-inv S and
    E  $\notin$  # learned-clss S and
    E  $\in$  # learned-clss T
  shows  $\exists S'. \text{backtrack } S S' \wedge \text{conflicting } S = \text{Some } E \wedge \text{full } cdcl_W\text{-cp } S' T$ 
  using assms
proof induction
  case conflict'
  then show ?case unfolding full1-def by (auto dest: rtrancpl-cdclW-cp-learned-clause-inv)
next
case (other' T U) note o = this(1) and cp = this(3) and not-yet = this(5) and learned = this(6)
  have E  $\in$  # learned-clss T
  using learned cp rtrancpl-cdclW-cp-learned-clause-inv unfolding full-def by auto
  then have backtrack S T and conflicting S = Some 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: raw-conflicting S = Some E and
    already-learned: mset-ccls E  $\in$  # 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 L M1 M2-loc K i where
    T: T ~ cons-trail (Propagated L (cls-of-ccls E))
    (reduce-trail-to M1 (add-learned-cls (cls-of-ccls E)
    (update-backtrack-lvl i (update-conflicting None S))))
    and
    decomp: (Marked K (Suc i) # M1, M2-loc) ∈
        set (get-all-marked-decomposition (trail S)) and
    LD: L ∈ # mset-ccls E and
    k: get-level (trail S) L = backtrack-lvl S and
    level: get-level (trail S) L = get-maximum-level (trail S) (mset-ccls E) and
    confl-S: raw-conflicting S = Some E and
    i: i = get-maximum-level (trail S) (remove1-mset L (mset-ccls E)) and
    undef: undefined-lit M1 L
    using confl by (induction rule: backtrack-induction-lev2) fastforce
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)
let ?E = mset-ccls E
let ?E' = remove1-mset L ?E
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 conf: cdclW-conflicting S unfolding cdclW-all-struct-inv-def by blast
then have trail S ⊨as CNot ?E unfolding cdclW-conflicting-def confl-S by auto
then have MD: trail S ⊨as CNot ?E by auto
then have MD': trail S ⊨as CNot ?E' using true-annot-CNot-remove1-mset-remove1-mset by blast
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 ?E' ⊆ atm-of ' lits-of-l 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-decomp)
have no-dup (trail S) using lev' by (auto simp: cdclW-M-level-inv-decomp)
have vars-in-M1:
    ∀ x ∈ atms-of ?E'. x ∉ atm-of ' lits-of-l (M2 @ [Marked K (i + 1)])
    unfolding Set.Ball-def apply (intro impI allI)
    apply (rule vars-of-D distinct-atms-of-incl-not-in-other[of
        M2 @ Marked K (i + 1) # [] M1 ?E'])
    using ⟨no-dup (trail S)⟩ M vars-of-D by simp-all
have M1-D: M1 ⊨as CNot ?E'
    using vars-in-M1 true-annots-remove-if-notin-vars[of M2 @ Marked K (i + 1) # [] M1 CNot ?E']
    MD' M by simp

have get-lvls-M: get-all-levels-of-marked (trail S) = rev [1..W-M-level-inv-def by auto
then have backtrack-lvl S > 0 unfolding M by (auto split: if-split-asm simp add: upt.simps(2))

obtain M1' K' Ls where
    M': trail S = Ls @ Marked K' (backtrack-lvl S) # M1' and
    Ls: ∀ l ∈ set Ls. ¬ is-marked l and

```

```

set M1 ⊆ set M1'
proof -
  let ?Ls = takeWhile (Not o is-marked) (trail S)
  have MLs: trail S = ?Ls @ dropWhile (Not o is-marked) (trail S)
    by auto
  have dropWhile (Not o is-marked) (trail S) ≠ [] unfolding M by auto
  moreover
    from hd-dropWhile[OF this] have is-marked(hd (dropWhile (Not o is-marked) (trail S)))
      by simp
  ultimately
    obtain K' K'k where
      K'k: dropWhile (Not o is-marked) (trail S)
        = Marked K' K'k # tl (dropWhile (Not o is-marked) (trail S))
      by (cases dropWhile (Not o is-marked) (trail S);
          cases hd (dropWhile (Not o is-marked) (trail S)))
          simp-all
  moreover have ∀ l ∈ set ?Ls. ¬is-marked l using set-takeWhileD by force
  moreover
    have get-all-levels-of-marked (trail S)
      = K'k # get-all-levels-of-marked(tl (dropWhile (Not o is-marked) (trail S)))
    apply (subst MLs, subst K'k)
    using calculation(2) by (auto simp add: get-all-levels-of-marked-no-marked)
    then have K'k = backtrack-lvl S
      using calculation(2) by (auto split: if-split-asm simp add: get-lvls-M upt.simps(2))
  moreover have set M1 ⊆ set (tl (dropWhile (Not o is-marked) (trail S)))
    unfolding M by (induction M2) auto
  ultimately show ?thesis using that MLs by metis
qed

have get-lvls-M: get-all-levels-of-marked (trail S) = rev [1..W-M-level-inv-def by auto
then have backtrack-lvl S > 0 unfolding M by (auto split: if-split-asm simp add: upt.simps(2) i)

have M1'-D: M1' ⊨as CNot ?E' using M1-D (set M1 ⊆ set M1') by (auto intro: true-annots-mono)
have -L ∈ lits-of-l (trail S) using conf confl-S LD unfolding cdclW-conflicting-def
  by (auto simp: in-CNot-implies-uminus)
have lvls-M1': get-all-levels-of-marked M1' = rev [1..W-stgy** R Y and
  YZ: cdclW-stgy Y Z and
  nt: ¬ (∃ c. trail Y = c @ Marked K' (backtrack-lvl S) # M1' @ []) and
  Z: (λa b. cdclW-stgy a b ∧ (∃ c. trail a = c @ Marked K' (backtrack-lvl S) # M1' @ []))** Z S
  using rtranclp-cdclW-new-marked-at-beginning-is-decide[OF st' - lev, of Ls K'
    backtrack-lvl S M1' []] unfolding R M' by auto

```

```

have [simp]: cdclW-M-level-inv Y
  using RY lev rtrancp-cdclW-stgy-consistent-inv by blast
obtain M' where trZ: trail Z = M' @ Marked K' (backtrack-lvl S) # M1'
  using rtrancp-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
have no-dup (trail Y)
  using RY lev rtrancp-cdclW-stgy-consistent-inv unfolding cdclW-M-level-inv-def by blast
then obtain Y' where
  dec: decide Y Y' and
  Y'Z: full cdclW-cp Y' Z and
  no-step cdclW-cp Y
  using cdclW-stgy-trail-has-new-marked-is-decide-step[OF YZ nt Z] M' by auto
have trY: trail Y = M1'
proof -
  obtain M' where M: trail Z = M' @ Marked K' (backtrack-lvl S) # M1'
    using rtrancp-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
  obtain M'' where M'': trail Z = M'' @ trail Y' and  $\forall m \in \text{set } M''. \neg \text{is-marked } m$ 
    using Y'Z rtrancp-cdclW-cp-dropWhile-trail' unfolding full-def by blast
  obtain M''' where trail Y' = M''' @ Marked K' (backtrack-lvl S) # M1'
    using M'' unfolding M
    by (metis (no-types, lifting)  $\forall m \in \text{set } M''. \neg \text{is-marked } m$  beginning-not-marked-invert)
  then show ?thesis using dec nt by (induction M'') (auto elim: decideE)
qed
have Y-CT: conflicting Y = None using  $\langle \text{decide } Y Y' \rangle$  by (auto elim: decideE)
have cdclW** R Y by (simp add: RY rtrancp-cdclW-stgy-rtrancp-cdclW)
then have init-clss Y = init-clss R using rtrancp-cdclW-init-clss[of R Y] M-lev by auto
{ assume DL: mset-ccls E  $\notin$  # clauses Y
  have atm-of L  $\notin$  atm-of ' lits-of-l M1
    apply (rule backtrack-lit-skipped[of S])
    using decomp i k lev' unfolding cdclW-M-level-inv-def by auto
  then have LM1: undefined-lit M1 L
    by (metis Marked-Propagated-in-iff-in-lits-of-l atm-of-uminus image-eqI)
  have L-trY: undefined-lit (trail Y) L
    using L-notin  $\langle \text{no-dup } (\text{trail } S) \rangle$  unfolding defined-lit-map trY M'
    by (auto simp add: image-iff lits-of-def)
  obtain E' where
    E': E' ! $\in$ ! raw-clauses Y and
    EE': mset-cls E' = mset-ccls E
    using DL in-mset-clss-exists-preimage by blast
  have Ex (propagate Y)
    using propagate-rule[of Y E' L] DL M1'-D L-trY Y-CT trY LD E'
    by (auto simp: EE')
  then have False using  $\langle \text{no-step cdcl}_W\text{-cp } Y \rangle$  propagate' by blast
}
moreover {
  assume DL: mset-ccls E  $\notin$  # clauses Y
  have lY-lZ: learned-clss Y = learned-clss Z
    using dec Y'Z rtrancp-cdclW-cp-learned-clause-inv[of Y' Z] unfolding full-def
    by (auto elim: decideE)
  have invZ: cdclW-all-struct-inv Z
    by (meson RY YZ invR r-into-rtrancp rtrancp-cdclW-all-struct-inv-inv
      rtrancp-cdclW-stgy-rtrancp-cdclW)
  have n: mset-ccls E  $\notin$  # learned-clss Z
    using DL lY-lZ YZ unfolding raw-clauses-def by auto
  have ?E  $\notin$  # learned-clss S
    apply (rule rtrancp-cdclW-stgy-with-trail-end-has-not-been-learned[OF Z invZ trZ])

```

```

    apply (simp add: n)
    using LD 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 rtranclp-cdclW-stgy-no-more-init-clss[of R S]
    unfolding M'
    by (simp add: ⟨init-clss Y = init-clss R⟩ raw-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 = Some E and
        cls-S': clauses S' = {#E#} + clauses S
        using ⟨cdclW-M-level-inv S⟩
        by (induction rule: backtrack-induction-lev2) (auto simp: cdclW-M-level-inv-decomp)
      then have E ∉ clauses S
        using cdclW-stgy-no-relearned-clause R invR local.backtrack st by blast
      then show ?thesis using IH by (simp add: distinct-mset-add-single cls-S')
    qed (auto elim: decideE skipE resolveE)
  qed
qed

```

qed

lemma *cdcl_W-stgy-distinct-mset-clauses*:

assumes
st: *cdcl_W-stgy*** (*init-state N*) *S* **and**
no-duplicate-clause: *distinct-mset* (*mset-clss N*) **and**
no-duplicate-in-clause: *distinct-mset-mset* (*mset-clss N*)
shows *distinct-mset* (*clauses S*)
using *rtrancp-cdcl_W-stgy-distinct-mset-clauses*[*OF - st*] *assms*
by (*auto simp: cdcl_W-all-struct-inv-def distinct-cdcl_W-state-def*)

18.8 Decrease of a measure

fun *cdcl_W-measure* **where**

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

lemma *length-model-le-vars-all-inv*:

assumes *cdcl_W-all-struct-inv S*
shows *length* (*trail S*) \leq *card* (*atms-of-mm* (*init-clss S*))
using *assms length-model-le-vars*[*of S*] **unfolding** *cdcl_W-all-struct-inv-def*
by (*auto simp: cdcl_W-M-level-inv-decomp*)

end

context *conflict-driven-clause-learning_W*

begin

lemma *learned-clss-less-upper-bound*:

fixes *S* :: '*st*'
assumes
distinct-cdcl_W-state S **and**
 $\forall s \in \# \text{learned-clss } S. \neg \text{tautology } s$
shows *card*(*set-mset* (*learned-clss S*)) $\leq \exists \wedge$ *card* (*atms-of-mm* (*learned-clss S*))

proof -

have *set-mset* (*learned-clss S*) \subseteq *simple-clss* (*atms-of-mm* (*learned-clss S*))
apply (*rule simplified-in-simple-clss*)
using *assms* **unfolding** *distinct-cdcl_W-state-def* **by** *auto*
then have *card*(*set-mset* (*learned-clss S*))
 \leq *card* (*simple-clss* (*atms-of-mm* (*learned-clss S*)))
by (*simp add: simple-clss-finite card-mono*)
then show *?thesis*
by (*meson atms-of-ms-finite simple-clss-card finite-set-mset order-trans*)

qed

lemma *lexn3*[*intro!*, *simp*]:

$a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$
 $\implies ([a :: \text{nat}, b, c], [a', b', c']) \in \text{lexn } \{(x, y). x < y\} \exists$
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:

$\neg(\text{learned-clss } S \subseteq \# \text{ learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = \text{None})$

and

no-forget: *learned-clss S* $\subseteq \#$ *learned-clss S'* **and**

no-relearn: $\bigwedge S'. \text{backtrack } S S' \implies \forall T. \text{conflicting } S = \text{Some } T \longrightarrow T \notin \# \text{learned-clss } S$

and

alien: *no-strange-atm S* **and**

M-level: *cdcl_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**

conf: *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** *conf = this(1)* **and** *undef = this(5)* **and** *T = this(6)*

have *propa*: *propagate S (cons-trail (Propagated L C) S)*

using *propagate-rule[OF propagate.hyps(1,2)] propagate.hyps* **by** *auto*

then have *no-dup'*: *no-dup (Propagated L (mset-cls C) # trail S)*

using *M-level cdcl_W-M-level-inv-decomp(2) undef defined-lit-map* **by** *auto*

let $?N = \text{init-clss } S$

have *no-strange-atm (cons-trail (Propagated L C) S)*

using *alien cdcl_W.propagate cdcl_W-no-strange-atm-inv propa M-level* **by** *blast*

then have *atm-of ' lits-of-l (Propagated L (mset-cls C) # trail S)*

$\subseteq \text{atms-of-mm (init-clss } S)$

using *undef unfolding no-strange-atm-def* **by** *auto*

then have *card (atm-of ' lits-of-l (Propagated L (mset-cls C) # trail S))*

$\leq \text{card (atms-of-mm (init-clss } S))$

by (*meson atms-of-ms-finite card-mono finite-set-mset*)

then have *length (Propagated L (mset-cls C) # trail S) ≤ card (atms-of-mm ?N)*

using *no-dup-length-eq-card-atm-of-lits-of-l no-dup'* **by** *fastforce*

then have *H*: *card (atms-of-mm (init-clss S)) − length (trail S)*

$= \text{Suc (card (atms-of-mm (init-clss } S)) - \text{Suc (length (trail } S))}$

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-rule decide.hyps* **by** *force*

then have *cdcl_W:cdcl_W S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))*

using *cdcl_W.simps cdcl_W-o.intros* **by** *blast*

moreover

have *lev*: *cdcl_W-M-level-inv (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))*

using *cdcl_W M-level cdcl_W-consistent-inv[OF cdcl_W]* **by** *auto*

then have *no-dup*: *no-dup (Marked L (backtrack-lvl S + 1) # trail S)*

using *undef unfolding cdcl_W-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) cdcl_W-no-strange-atm-inv* **by** *blast*

then have *length (Marked L ((backtrack-lvl S) + 1) # (trail S))*

```

    ≤ card (atms-of-mm (init-clss S))
    using no-dup 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 by (simp add: tr)
next
  case conflict
  then show ?case by simp
next
  case resolve
  then show ?case using finite by simp
next
  case (backtrack K i M1 M2 L D T) note conf = this(1) and decomp = this(3) and undef = this(7)
  and
    T = this(8) and lev = this(9)
  let ?S' = T
  have bt: backtrack S ?S'
    using backtrack.hyps backtrack.intros[of S D L K i] by auto
  have mset-ccls D ∉ # learned-clss S
    using no-relearn conf bt by auto
  then have card-T:
    card (set-mset ({#mset-ccls D#} + learned-clss S)) = Suc (card (set-mset (learned-clss S)))
    by simp
  have distinct-cdclW-state ?S'
    using bt M-level distinct-cdclW-state-inv no-dup other cdclW-o.intros cdclW-bj.intros by blast
  moreover have ∀ s ∈ # learned-clss ?S'. ¬ tautology s
    using learned-clss-are-not-tautologies[OF cdclW.other[OF cdclW-o.bj[OF
      cdclW-bj.backtrack[OF bt]]]] M-level no-taut confl by auto
  ultimately have card (set-mset (learned-clss T)) ≤ 3 ^ card (atms-of-mm (learned-clss T))
    by (auto simp: learned-clss-less-upper-bound)
  then have H: card (set-mset ({#mset-ccls D#} + learned-clss S))
    ≤ 3 ^ card (atms-of-mm ({#mset-ccls D#} + learned-clss S))
    using T undef decomp M-level by (simp add: cdclW-M-level-inv-decomp)
  moreover
    have atms-of-mm ({#mset-ccls D#} + learned-clss S) ⊆ atms-of-mm (init-clss S)
      using alien conf unfolding no-strange-atm-def by auto
    then have card-f: card (atms-of-mm ({#mset-ccls D#} + learned-clss S))
      ≤ card (atms-of-mm (init-clss S))
      by (meson atms-of-mm-finite card-mono finite-set-mset)
    then have (3::nat) ^ card (atms-of-mm ({#mset-ccls D#} + learned-clss S))
      ≤ 3 ^ card (atms-of-mm (init-clss S)) by simp
  ultimately have (3::nat) ^ card (atms-of-mm (init-clss S))
    ≥ card (set-mset ({#mset-ccls D#} + learned-clss S))
    using le-trans by blast
  then show ?case using decomp undef diff-less-mono2 card-T T M-level
    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) note no-forget = this(8)
  then have mset-cl C ∈ # learned-clss S and mset-cl C ∉ # learned-clss T

```

```

    using forget.hyps by auto
  then show ?case using no-forget by (auto 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) ∈ 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) ∈ lexn {(a, b). a < b} 3
  apply (rule cdclW-measure-decreasing)
  using assms(1) conflict apply blast
    using assms(1) apply (auto simp: state-eq-def simp del: state-simp elim!: conflictE)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def elim: conflictE)
  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: state-eq-def simp del: state-simp elim!: decideE)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def elim: decideE)
  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) ∈ lexn {a. case a of (a, b) ⇒ a < b} 3 ∨
      cdclW-measure U = cdclW-measure T
      using cp unfolding full-def rtranclp-unfold by blast
    moreover
      have cdclW-M-level-inv S
        using cdclW-all-struct-inv-def other'.prems(4) by blast
      with st have (cdclW-measure T, cdclW-measure S) ∈ lexn {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 decide.intros[OF decide.hyps] by blast
      next
        case (backtrack K i M1 M2 L D T) note conf = this(1) and decomp = this(3) and
          undef = this(7) and T = this(8)
        have bt: backtrack S T
          apply (rule backtrack-rule)
          using backtrack.hyps by auto
        then have no-relearn: ∀ T. conflicting S = Some T → T ∉ learned-clss S

```

```

    using cdclW-stgy-no-relearned-clause[of R S T] H conf
    unfolding cdclW-all-struct-inv-def raw-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 leW-transI transD trans-le)
qed
qed

```

lemma *tranclp-cdcl_W-stgy-decreasing*:

```

  fixes R S T :: 'st
  assumes cdclW-stgy++ R S
  trail R = [] and
  cdclW-all-struct-inv R
  shows (cdclW-measure S, cdclW-measure R) ∈ leW {(a, b). a < b} 3
  using assms
  apply induction
    using cdclW-stgy-step-decreasing[of R - R] apply blast
  using cdclW-stgy-step-decreasing[of - - R] tranclp-into-rtranclp[of cdclW-stgy R]
  leW-transI[OF trans-le, of 3] unfolding trans-def by blast

```

lemma *tranclp-cdcl_W-stgy-S0-decreasing*:

```

  fixes R S T :: 'st
  assumes
    pl: cdclW-stgy++ (init-state N) S and
    no-dup: distinct-mset-mset (mset-clss N)
  shows (cdclW-measure S, cdclW-measure (init-state N)) ∈ leW {(a, b). a < b} 3
proof -
  have cdclW-all-struct-inv (init-state N)
    using no-dup unfolding cdclW-all-struct-inv-def by auto
  then show ?thesis using pl tranclp-cdclW-stgy-decreasing init-state-trail by blast
qed

```

lemma *wf-tranclp-cdcl_W-stgy*:

```

  wf {(S :: 'st, init-state N)|

```

```

    S N. distinct-mset-mset (mset-cls N) ∧ cdclW-stgy++ (init-state N) S}
apply (rule wf-wf-if-measure'-notation2[of lexn {(a, b). a < b} 3 - - cdclW-measure])
apply (simp add: wf wf-lexn)
using tranclp-cdclW-stgy-S0-decreasing by blast

lemma cdclW-cp-wf-all-inv:
  wf {(S', S). cdclW-all-struct-inv S ∧ cdclW-cp S S'}
  (is wf ?R)
proof (rule wf-bounded-measure[of -
  λS. card (atms-of-mm (init-cls S))+1
  λS. length (trail S) + (if conflicting S = None then 0 else 1)], goal-cases)
case (1 S S')
then have cdclW-all-struct-inv S and cdclW-cp S S' by auto
moreover then have cdclW-all-struct-inv S'
  using cdclW-cp.simps cdclW-all-struct-inv-inv conflict cdclW.intros cdclW-all-struct-inv-inv
  by blast+
ultimately show ?case
  by (auto simp:cdclW-cp.simps state-eq-def simp del: state-simp elim!: conflictE propagateE
    dest: length-model-le-vars-all-inv)
qed

end

end
theory DPLL-CDCL-W-Implementation
imports Partial-Annotated-Clausal-Logic
begin

```

19 Simple Implementation of the DPLL and CDCL

19.1 Common Rules

19.1.1 Propagation

The following theorem holds:

```

lemma lits-of-l-unfold[iff]:
  (∀ c ∈ set C. -c ∈ lits-of-l Ms) ↔ Ms ⊨as CNot (mset C)
  unfolding true-annot-def Ball-def true-annot-def CNot-def by auto

```

The right-hand version is written at a high-level, but only the left-hand side is executable.

```

definition is-unit-clause :: 'a literal list ⇒ ('a, 'b, 'c) marked-lit list ⇒ 'a literal option
where
  is-unit-clause l M =
    (case List.filter (λa. atm-of a ∉ atm-of ' lits-of-l M) l of
      a # [] ⇒ if M ⊨as CNot (mset l - {#a#}) then Some a else None
      | - ⇒ None)

```

```

definition is-unit-clause-code :: 'a literal list ⇒ ('a, 'b, 'c) marked-lit list
  ⇒ 'a literal option where
  is-unit-clause-code l M =
    (case List.filter (λa. atm-of a ∉ atm-of ' lits-of-l M) l of
      a # [] ⇒ if (∀ c ∈ set (remove1 a l). -c ∈ lits-of-l M) then Some a else None
      | - ⇒ None)

```

```

lemma is-unit-clause-is-unit-clause-code[code]:

```

```

is-unit-clause l M = is-unit-clause-code l M
proof -
  have 1:  $\bigwedge a. (\forall c \in \text{set } (\text{remove1 } a \ l). - c \in \text{lits-of-}l \ M) \longleftrightarrow M \models_{as} CNot \ (mset \ l - \ \{\#a\# \})$ 
    using lits-of-l-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 ← l . atm-of a ∉ atm-of ‘ lits-of-l M] of [] ⇒ None
    | [a] ⇒ if M ⊨as CNot (mset l - {#a#}) then Some a else None
    | a # ab # xa ⇒ Map.empty xa) = Some a
    using assms unfolding is-unit-clause-def .
  hence a ∈ set [a ← l . atm-of a ∉ atm-of ‘ lits-of-l M]
  apply (cases [a ← l . atm-of a ∉ atm-of ‘ lits-of-l M])
  apply simp
  apply (rename-tac aa list; case-tac list) by (auto split: if-split-asm)
  hence atm-of a ∉ atm-of ‘ lits-of-l M by auto
  thus ?thesis
    by (simp add: Marked-Propagated-in-iff-in-lits-of-l
      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 ⟹ M ⊨as CNot (mset l - {#a#})
  unfolding is-unit-clause-def
proof -
  assume (case [a ← l . atm-of a ∉ atm-of ‘ lits-of-l M] of [] ⇒ None
    | [a] ⇒ if M ⊨as CNot (mset l - {#a#}) then Some a else None
    | a # ab # xa ⇒ Map.empty xa) = Some a
  thus ?thesis
    apply (cases [a ← l . atm-of a ∉ atm-of ‘ lits-of-l M], simp)
    apply simp
    apply (rename-tac aa list; case-tac list) by (auto split: if-split-asm)
qed

lemma is-unit-clause-some-in: is-unit-clause l M = Some a ⟹ a ∈ set l
  unfolding is-unit-clause-def
proof -
  assume (case [a ← l . atm-of a ∉ atm-of ‘ lits-of-l M] of [] ⇒ None
    | [a] ⇒ if M ⊨as CNot (mset l - {#a#}) then Some a else None
    | a # ab # xa ⇒ Map.empty xa) = Some a
  thus a ∈ set l
    by (cases [a ← l . atm-of a ∉ atm-of ‘ lits-of-l M])
      (fastforce dest: filter-eq-ConsD split: if-split-asm split: list.splits)+
qed

lemma is-unit-clause-nil[simp]: is-unit-clause [] M = None
  unfolding is-unit-clause-def by auto

```

19.1.2 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  $\notin$  atm-of ' lits-of-l 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-l-unfold
      simp add: lits-of-l-unfold[symmetric] Marked-Propagated-in-iff-in-lits-of-l
      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 +  $\{\#ac\# \} - \{\#a\# \} = \text{mset } c - \{\#a\# \} + \{\#ac\# \}$ 
  by (auto simp add: multiset-eq-iff)
  show ?thesis using False Cons
  by (auto simp add: T atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
  qed
qed
thus ?thesis
using M unfolding is-unit-clause-def by auto
qed

```

```

lemma find-first-unit-clause-none:
distinct c  $\implies c \in \text{set } l \implies M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \}) \implies \text{undefined-lit } M a \implies a \in \text{set } c$ 
 $\implies \text{find-first-unit-clause } l M \neq \text{None}$ 
by (induction l)
  (auto split: option.split simp add: propagate-is-unit-clause-not-None)

```

19.1.3 Decide

fun *find-first-unused-var* :: 'a literal list list \Rightarrow 'a literal set \Rightarrow 'a literal option **where**

find-first-unused-var (*a* # *l*) *M* =
 (case *List.find* ($\lambda \text{lit. lit} \notin M \wedge \neg \text{lit} \notin M$) *a* of
 None \Rightarrow *find-first-unused-var* *l* *M*
 | Some *a* \Rightarrow Some *a*) |
find-first-unused-var [] - = None

lemma *find-none*[*iff*]:

List.find ($\lambda \text{lit. lit} \notin M \wedge \neg \text{lit} \notin M$) *a* = None \longleftrightarrow atm-of ' set *a* \subseteq 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* ($\lambda \text{lit. lit} \notin M \wedge \neg \text{lit} \notin M$) *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-l* *Ms*) = Some *a* $\implies \text{undefined-lit } Ms \ a$
using *find-first-unused-var-Some*[of *l lits-of-l Ms a*] *Marked-Propagated-in-iff-in-lits-of-l*
by *blast*

end

theory *DPLL-W-Implementation*

imports *DPLL-CDCL-W-Implementation DPLL-W* $\sim\sim$ /src/HOL/Library/Code-Target-Numeral
begin

19.2 Simple Implementation of DPLL

19.2.1 Combining the propagate and decide: a DPLL step

definition *DPLL-step* :: int *dpll_W-marked-lits* \times int literal list list

\Rightarrow int *dpll_W-marked-lits* \times int literal list list **where**

DPLL-step = ($\lambda(Ms, N).$

(case *find-first-unit-clause* *N* *Ms* of
 Some (*L*, -) \Rightarrow (*Propagated L* ()) # *Ms*, *N*)

```

| - =>
  if  $\exists C \in \text{set } N. (\forall c \in \text{set } C. -c \in \text{lits-of-l } Ms)$ 
  then
    (case backtrack-split Ms of
      (-, L # M) => (Propagated (- (lit-of L)) () # M, N)
    | (-, -) => (Ms, N)
    )
  else
    (case find-first-unused-var N (lits-of-l Ms) of
      Some a => (Marked a () # Ms, N)
    | None => (Ms, N)))

```

Example of propagation:

```

value DPLL-step ([Marked (Neg 1) ()], [[Pos (1::int), Neg 2]])

```

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

```

abbreviation toS  $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list})$ 
  ( $N:: \text{int literal list list}$ ). (Ms, mset (map mset N))
abbreviation toS'  $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list},$ 
   $N:: \text{int literal list list})$ . (Ms, mset (map mset N))

```

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-dpll_W-step*:

```

assumes step: (Ms', N') = DPLL-step (Ms, N)
and neg: (Ms, N)  $\neq$  (Ms', N')
shows dpllW (toS Ms N) (toS Ms' N')

```

proof –

```

let ?S = (Ms, mset (map mset N))
{ fix L E
  assume unit: find-first-unit-clause N Ms = Some (L, E)
  hence Ms'N: (Ms', N') = (Propagated L () # Ms, N)
    using step unfolding DPLL-step-def by auto
  obtain C where
    C: C  $\in$  set N and
    Ms: Ms  $\models_{\text{as}}$  CNot (mset C - {#L#}) and
    undef: undefined-lit Ms L and
    L  $\in$  set C using find-first-unit-clause-some[OF unit] by metis
  have dpllW (Ms, mset (map mset N))
    (Propagated L () # fst (Ms, mset (map mset N)), snd (Ms, mset (map mset N)))
    apply (rule dpllW.propagate)
    using Ms undef C (L  $\in$  set C) by (auto simp add: C)
  hence ?thesis using Ms'N by auto
}

```

moreover

```

{ assume unit: find-first-unit-clause N Ms = None
  assume exC:  $\exists C \in \text{set } N. Ms \models_{\text{as}}$  CNot (mset C)
  then obtain C where C: C  $\in$  set N and Ms: Ms  $\models_{\text{as}}$  CNot (mset C) by auto
  then obtain L M M' where bt: backtrack-split Ms = (M', L # M)
    using step exC neg unfolding DPLL-step-def prod.case unit
    by (cases backtrack-split Ms, rename-tac b, case-tac b) auto
  hence is-marked L using backtrack-split-snd-hd-marked[of Ms] by auto
  have 1: dpllW (Ms, mset (map mset N))
    (Propagated (- lit-of L) () # M, snd (Ms, mset (map mset N)))
    apply (rule dpllW.backtrack[OF - (is-marked L), of ])

```

```

    using C Ms bt by auto
  moreover have (Ms', N') = (Propagated (¬ (lit-of L)) () # M, N)
    using step exC unfolding DPLL-step-def bt prod.case unit by auto
  ultimately have ?thesis by auto
}
moreover
{ assume unit: find-first-unit-clause N Ms = None
  assume exC: ¬ (∃ C ∈ set N. Ms ⊨as CNot (mset C))
  obtain L where unused: find-first-unused-var N (lits-of-l Ms) = Some L
    using step exC neq unfolding DPLL-step-def prod.case unit
    by (cases find-first-unused-var N (lits-of-l Ms)) auto
  have dpllW (Ms, mset (map mset N))
    (Marked L () # fst (Ms, mset (map mset N)), snd (Ms, mset (map mset N)))
    apply (rule dpllW.decided[of ?S L])
    using find-first-unused-var-Some[OF unused]
    by (auto simp add: Marked-Propagated-in-iff-in-lits-of-l atms-of-ms-def)
  moreover have (Ms', N') = (Marked L () # Ms, N)
    using step exC unfolding DPLL-step-def unused prod.case unit by auto
  ultimately have ?thesis by auto
}
ultimately show ?thesis by (cases find-first-unit-clause N Ms) auto
qed

lemma DPLL-step-stuck-final-state:
  assumes step: (Ms, N) = DPLL-step (Ms, N)
  shows conclusive-dpllW-state (toS Ms N)
proof -
  have unit: find-first-unit-clause N Ms = None
    using step unfolding DPLL-step-def by (auto split: option.splits)

  { assume n: ∃ C ∈ set N. Ms ⊨as CNot (mset C)
    hence Ms: (Ms, N) = (case backtrack-split Ms of (x, []) ⇒ (Ms, N)
      | (x, L # M) ⇒ (Propagated (¬ lit-of L) () # M, N))
      using step unfolding DPLL-step-def by (simp add: unit)

  have snd (backtrack-split Ms) = []
  proof (cases backtrack-split Ms, cases snd (backtrack-split Ms))
    fix a b
    assume backtrack-split Ms = (a, b) and snd (backtrack-split Ms) = []
    thus snd (backtrack-split Ms) = [] by blast
  next
    fix a b aa list
    assume
      bt: backtrack-split Ms = (a, b) and
      bt': snd (backtrack-split Ms) = aa # list
    hence Ms: Ms = Propagated (¬ lit-of aa) () # list using Ms by auto
    have is-marked aa using backtrack-split-snd-hd-marked[of Ms] bt bt' by auto
    moreover have fst (backtrack-split Ms) @ aa # list = Ms
      using backtrack-split-list-eq[of Ms] bt' by auto
    ultimately have False unfolding Ms by auto
    thus snd (backtrack-split Ms) = [] by blast
  qed

  hence ?thesis
    using n backtrack-snd-empty-not-marked[of Ms] unfolding conclusive-dpllW-state-def

```



```

    by (cases backtrack-split Ms) auto
  }
  moreover {
    assume n:  $\neg (\exists C \in \text{set } N. Ms \models_{\text{as}} \text{CNot } (\text{mset } C))$ 
    hence find-first-unused-var N (lits-of-l Ms) = None
    using step unfolding DPLL-step-def by (simp add: unit split: option.splits)
    hence a:  $\forall a \in \text{set } N. \text{atm-of } ' \text{set } a \subseteq \text{atm-of } ' (\text{lits-of-l Ms})$  by auto
    have fst (toS Ms N)  $\models_{\text{asm}} \text{snd } (toS Ms N)$  unfolding true-annot-def CNot-def Ball-def
    proof clarify
      fix x
      assume x:  $x \in \text{set-mset } (\text{clauses } (toS Ms N))$ 
      hence  $\neg Ms \models_{\text{as}} \text{CNot } x$  using n unfolding true-annot-def CNot-def Ball-def by auto
      moreover have total-over-m (lits-of-l 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)  $\models_a x$ 
        using total-not-CNot[of lits-of-l Ms x] by (simp add: true-annot-def true-annot-true-cls)
    qed
    hence ?thesis unfolding conclusive-dpllW-state-def by blast
  }
  ultimately show ?thesis by blast
qed

```

19.2.2 Adding invariants

Invariant tested in the function `function DPLL-ci :: int dpllW-marked-lits \Rightarrow int literal list list`

```

 $\Rightarrow$  int dpllW-marked-lits  $\times$  int literal list list where
DPLL-ci Ms N =
  (if  $\neg \text{dpll}_W\text{-all-inv } (Ms, \text{mset } (\text{map } \text{mset } N))$ 
   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)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  dpllW S S'}})
  show wf {(S', S). (toS' S', toS' S)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  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  $\neg \neg \text{dpll}_W\text{-all-inv } (toS Ms N)$ 
  and step: x = DPLL-step (Ms, N)
  and x: (xa, y) = x
  and (xa, y)  $\neq$  (Ms, N)
  thus ((xa, N), Ms, N)  $\in$  {(S', S). (toS' S', toS' S)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  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 dpllW-marked-lits \Rightarrow int literal list list`

```

 $\Rightarrow$ 
  int dpllW-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: if-split option.splits prod.splits list.splits)

lemma dpllW-all-inv-implicS-2-eq3-and-dom:
  assumes dpllW-all-inv (Ms, mset (map mset N))
  shows DPLL-ci Ms N = DPLL-part Ms N ∧ 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 (cases DPLL-step (Ms, N)) auto
  have inv': dpllW-all-inv (toS Ms' N) by (metis (mono-tags) 1.prem DPLL-step-is-a-dpllW-step
    Ms' dpllW-all-inv old.prod.inject)
  { assume (Ms', N) ≠ (Ms, N)
    hence DPLL-ci Ms' N = DPLL-part Ms' N ∧ DPLL-part-dom (Ms', N) using 1(1)[of - Ms' N]
  }
  Ms'
  1(2) inv' by auto
  hence DPLL-part-dom (Ms, N) using DPLL-part.domintros Ms' by fastforce
  moreover have DPLL-ci Ms N = DPLL-part Ms N using 1.prem DPLL-part.psimps Ms'
    ⟨DPLL-ci Ms' N = DPLL-part Ms' N ∧ DPLL-part-dom (Ms', N)⟩ ⟨DPLL-part-dom (Ms, N)⟩ by
auto
    ultimately have ?case by blast
  }
  moreover {
    assume (Ms', N) = (Ms, N)
    hence ?case using DPLL-part.domintros DPLL-part.psimps Ms' by fastforce
  }
  ultimately show ?case by blast
qed

lemma DPLL-ci-dpllW-rtrancp:
  assumes DPLL-ci Ms N = (Ms', N')
  shows dpllW** (toS Ms N) (toS Ms' N')
  using assms
proof (induct Ms N arbitrary: Ms' N' rule: DPLL-ci.induct)
  case (1 Ms N Ms' N') note IH = this(1) and step = this(2)
  obtain S1 S2 where S: (S1, S2) = DPLL-step (Ms, N) by (cases DPLL-step (Ms, N)) auto

  { assume ¬dpllW-all-inv (toS Ms N)
    hence (Ms, N) = (Ms', N) using step by auto
    hence ?case by auto
  }
  moreover
  { assume dpllW-all-inv (toS Ms N)
    and (S1, S2) = (Ms, N)
    hence ?case using S step by auto
  }
  moreover
  { assume dpllW-all-inv (toS Ms N)
    and (S1, S2) ≠ (Ms, N)
    moreover obtain S1' S2' where DPLL-ci S1 N = (S1', S2') by (cases DPLL-ci S1 N) auto
    moreover have DPLL-ci Ms N = DPLL-ci S1 N using DPLL-ci.simps[of Ms N] calculation
    proof –
  }

```

```

have (case (S1, S2) of (ms, lss) ⇒
  if (ms, lss) = (Ms, N) then (Ms, N) else DPLL-ci ms N) = DPLL-ci Ms N
  using S DPLL-ci.simps[of Ms N] calculation by presburger
hence (if (S1, S2) = (Ms, N) then (Ms, N) else DPLL-ci S1 N) = DPLL-ci Ms N
  by fastforce
thus ?thesis
  using calculation(2) by presburger
qed
ultimately have dpllW** (toS S1' N) (toS Ms' N) using IH[of (S1, S2) S1 S2] S step by simp

moreover have dpllW (toS Ms N) (toS S1 N)
  by (metis DPLL-step-is-a-dpllW-step S ⟨S1, S2⟩ ≠ (Ms, N) prod.sel(2) snd-DPLL-step)
ultimately have ?case by (metis (mono-tags, hide-lams) IH S ⟨S1, S2⟩ ≠ (Ms, N)
  ⟨DPLL-ci Ms N = DPLL-ci S1 N⟩ ⟨dpllW-all-inv (toS Ms N)⟩ converse-rtrancp-into-rtrancp
  local.step)
}
ultimately show ?case by blast
qed

lemma dpllW-all-inv-dpllW-trancp-irrefl:
  assumes dpllW-all-inv (Ms, N)
  and dpllW++ (Ms, N) (Ms, N)
  shows False
proof –
  have 1: wf {(S', S). dpllW-all-inv S ∧ dpllW++ S S'} using dpllW-wf-trancp by auto
  have ((Ms, N), (Ms, N)) ∈ {(S', S). dpllW-all-inv S ∧ dpllW++ S S'} using assms by auto
  thus False using wf-not-refl[OF 1] by blast
qed

lemma DPLL-ci-final-state:
  assumes step: DPLL-ci Ms N = (Ms, N)
  and inv: dpllW-all-inv (toS Ms N)
  shows conclusive-dpllW-state (toS Ms N)
proof –
  have st: dpllW** (toS Ms N) (toS Ms N) using DPLL-ci-dpllW-rtrancp[OF step] .
  have DPLL-step (Ms, N) = (Ms, N)
  proof (rule ccontr)
    obtain Ms' N' where Ms'N: (Ms', N') = DPLL-step (Ms, N)
    by (cases DPLL-step (Ms, N)) auto
    assume ¬ ?thesis
    hence DPLL-ci Ms' N = (Ms, N) using step inv st Ms'N[symmetric] by fastforce
    hence dpllW++ (toS Ms N) (toS Ms N)
    by (metis DPLL-ci-dpllW-rtrancp DPLL-step-is-a-dpllW-step Ms'N ⟨DPLL-step (Ms, N) ≠ (Ms,
N)⟩
      prod.sel(2) rtrancp-into-trancp2 snd-DPLL-step)
    thus False using dpllW-all-inv-dpllW-trancp-irrefl inv by auto
  qed
  thus ?thesis using DPLL-step-stuck-final-state[of Ms N] by simp
qed

lemma DPLL-step-obtains:
  obtains Ms' where (Ms', N) = DPLL-step (Ms, N)
  unfolding DPLL-step-def by (metis (no-types, lifting) DPLL-step-def prod.collapse snd-DPLL-step)

lemma DPLL-ci-obtains:

```

obtains Ms' **where** $(Ms', N) = DPLL\text{-}ci\ Ms\ N$
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** $DPLL\text{-}step\text{-}obtains$ **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 $DPLL\text{-}ci\text{-}no\text{-}more\text{-}step$:

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** $DPLL\text{-}step\text{-}obtains$ **by** $auto$
{ assume $\neg dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
hence $?case$ **using** $step$ **by** $auto$
}
moreover **{**
assume $dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
and $(S_1, N) = (Ms, N)$
hence $?case$ **using** $S\ step$ **by** $auto$
}
moreover
{ assume $inv: dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
assume $n: (S_1, N) \neq (Ms, N)$
obtain S_1' **where** $SS: (S_1', N) = DPLL\text{-}ci\ S_1\ N$ **using** $DPLL\text{-}ci\text{-}obtains$ **by** $blast$
moreover have $DPLL\text{-}ci\ Ms\ N = DPLL\text{-}ci\ S_1\ N$
proof –
have $(case\ (S_1, N)\ of\ (ms, lss) \Rightarrow if\ (ms, lss) = (Ms, N)\ then\ (Ms, N)\ else\ DPLL\text{-}ci\ ms\ N)$
= $DPLL\text{-}ci\ Ms\ N$
using $S\ DPLL\text{-}ci.simps[of\ Ms\ N]$ $calculation\ inv$ **by** $presburger$
hence $(if\ (S_1, N) = (Ms, N)\ then\ (Ms, N)\ else\ DPLL\text{-}ci\ S_1\ N) = DPLL\text{-}ci\ Ms\ N$
by $fastforce$
thus $?thesis$
using $calculation\ n$ **by** $presburger$
qed
moreover
have $DPLL\text{-}ci\ S_1'\ N = (S_1', N)$ **using** $step\ IH[OF\ -\ -\ S\ n\ SS[symmetric]]\ inv$ **by** $blast$
ultimately have $?case$ **using** $step$ **by** $fastforce$

```

}
ultimately show ?case by blast
qed

```

lemma *DPLL-part-dpll_W-all-inv-final*:

fixes *M Ms':: (int, unit, unit) marked-lit list* **and**

N :: int literal list list

assumes *inv: dpll_W-all-inv (Ms, mset (map mset N))*

and *MsN: DPLL-part Ms N = (Ms', N)*

shows *conclusive-dpll_W-state (toS Ms' N) ∧ dpll_W** (toS Ms N) (toS Ms' N)*

proof –

have *2: DPLL-ci Ms N = DPLL-part Ms N* **using** *inv dpll_W-all-inv-implicS-2-eq3-and-dom* **by** *blast*

hence *star: dpll_W** (toS Ms N) (toS Ms' N)* **unfolding** *MsN* **using** *DPLL-ci-dpll_W-rtrancp* **by**

blast

hence *inv': dpll_W-all-inv (toS Ms' N)* **using** *inv rtrancp-dpll_W-all-inv* **by** *blast*

show *?thesis* **using** *star DPLL-ci-final-state[OF DPLL-ci-no-more-step inv'] 2* **unfolding** *MsN* **by**

blast

qed

Embedding the invariant into the type

Defining the type **typedef** *dpll_W-state* =

$\{(M::(int, unit, unit) \text{ marked-lit list}, N::int \text{ literal list list}).$

$dpll_W\text{-all-inv (toS M N)}\}$

morphisms *rough-state-of state-of*

proof

show $([], []) \in \{(M, N). dpll_W\text{-all-inv (toS M N)}\}$ **by** *(auto simp add: dpll_W-all-inv-def)*

qed

lemma

DPLL-part-dom ([], N)

using *assms dpll_W-all-inv-implicS-2-eq3-and-dom[of [] N]* **by** *(simp add: dpll_W-all-inv-def)*

Some type classes **instantiation** *dpll_W-state :: equal*

begin

definition *equal-dpll_W-state :: dpll_W-state \Rightarrow dpll_W-state \Rightarrow bool* **where**

equal-dpll_W-state S S' = (rough-state-of S = rough-state-of S')

instance

by *standard (simp add: rough-state-of-inject equal-dpll_W-state-def)*

end

DPLL **definition** *DPLL-step' :: dpll_W-state \Rightarrow dpll_W-state* **where**

DPLL-step' S = state-of (DPLL-step (rough-state-of S))

declare *rough-state-of-inverse[simp]*

lemma *DPLL-step-dpll_W-conc-inv:*

DPLL-step (rough-state-of S) $\in \{(M, N). dpll_W\text{-all-inv (toS M N)}\}$

by *(smt DPLL-ci.simps DPLL-ci-dpll_W-rtrancp case-prodE case-prodI2 rough-state-of mem-Collect-eq old.prod.case prod.sel(2) rtrancp-dpll_W-all-inv snd-DPLL-step)*

lemma *rough-state-of-DPLL-step'-DPLL-step[simp]:*

rough-state-of (DPLL-step' S) = DPLL-step (rough-state-of S)

using *DPLL-step-dpll_W-conc-inv DPLL-step'-def state-of-inverse* **by** *auto*

```

function DPLL-tot:: dpllW-state  $\Rightarrow$  dpllW-state where
DPLL-tot S =
  (let S' = DPLL-step' S in
    if S' = S then S else DPLL-tot S')
  by fast+
termination
proof (relation {(T', T).
  (rough-state-of T', rough-state-of T)
     $\in$  {(S', S). (toS' S', toS' S)
       $\in$  {(S', S). dpllW-all-inv S  $\wedge$  dpllW S S'}}})
show wf {(b, a).
  (rough-state-of b, rough-state-of a)
     $\in$  {(b, a). (toS' b, toS' a)
       $\in$  {(b, a). dpllW-all-inv a  $\wedge$  dpllW a b}}})
  using wf-if-measure-f[OF wf-if-measure-f[OF dpllW-wf, of toS'], of rough-state-of] .
next
fix S x
assume x: x = DPLL-step' S
and x  $\neq$  S
have dpllW-all-inv (case rough-state-of S of (Ms, N)  $\Rightarrow$  (Ms, mset (map mset N)))
  by (metis (no-types, lifting) case-prodE mem-Collect-eq old.prod.case rough-state-of)
moreover have dpllW (case rough-state-of S of (Ms, N)  $\Rightarrow$  (Ms, mset (map mset N)))
  (case rough-state-of (DPLL-step' S) of (Ms, N)  $\Rightarrow$  (Ms, mset (map mset N)))
proof –
  obtain Ms N where Ms: (Ms, N) = rough-state-of S by (cases rough-state-of S) auto
  have dpllW-all-inv (toS' (Ms, N)) using calculation unfolding Ms by blast
  moreover obtain Ms' N' where Ms': (Ms', N') = rough-state-of (DPLL-step' S)
    by (cases rough-state-of (DPLL-step' S)) auto
  ultimately have dpllW-all-inv (toS' (Ms', N')) unfolding Ms'
    by (metis (no-types, lifting) case-prod-unfold mem-Collect-eq rough-state-of)

  have dpllW (toS Ms N) (toS Ms' N')
    apply (rule DPLL-step-is-a-dpllW-step[of Ms' N' Ms N])
    unfolding Ms Ms' using  $\langle x \neq S \rangle$  rough-state-of-inject x by fastforce+
    thus ?thesis unfolding Ms[symmetric] Ms'[symmetric] by auto
qed
ultimately show (x, S)  $\in$  {(T', T). (rough-state-of T', rough-state-of T)
   $\in$  {(S', S). (toS' S', toS' S)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  dpllW S S'}}})
  by (auto simp add: x)
qed

lemma [code]:
DPLL-tot S =
  (let S' = DPLL-step' S in
    if S' = S then S else DPLL-tot S') by auto

lemma DPLL-tot-DPLL-step-DPLL-tot[simp]: DPLL-tot (DPLL-step' S) = DPLL-tot S
  apply (cases DPLL-step' S = S)
  apply simp
  unfolding DPLL-tot.simps[of S] by (simp del: DPLL-tot.simps)

lemma DOPLL-step'-DPLL-tot[simp]:
  DPLL-step' (DPLL-tot S) = DPLL-tot S
  by (rule DPLL-tot.induct[of  $\lambda S. \textit{DPLL-step'} (\textit{DPLL-tot } S) = \textit{DPLL-tot } S$ ])

```

(metis (full-types) DPLL-tot.simps)

lemma *DPLL-tot-final-state*:

assumes *DPLL-tot* $S = S$

shows *conclusive-dpll_W-state* (*toS'* (*rough-state-of* S))

proof –

have *DPLL-step'* $S = S$ **using** *assms[symmetric]* *DOPLL-step'-DPLL-tot* **by** *metis*

hence *DPLL-step* (*rough-state-of* S) = (*rough-state-of* S)

unfolding *DPLL-step'-def* **using** *DPLL-step-dpll_W-conc-inv* *rough-state-of-inverse*

by (*metis* *rough-state-of-DPLL-step'-DPLL-step*)

thus *?thesis*

by (*metis* (*mono-tags*, *lifting*) *DPLL-step-stuck-final-state* *old.prod.exhaust* *split-conv*)

qed

lemma *DPLL-tot-star*:

assumes *rough-state-of* (*DPLL-tot* S) = S'

shows *dpll_W*** (*toS'* (*rough-state-of* S)) (*toS'* S')

using *assms*

proof (*induction arbitrary: S' rule: DPLL-tot.induct*)

case (1 $S S'$)

let $?x = \text{DPLL-step}' S$

{ assume $?x = S$

then have $?case$ **using** 1(2) **by** *simp*

}

moreover {

assume $S: ?x \neq S$

have $?case$

apply (*cases* *DPLL-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-state-of-rough-state-of-nil[simp]*:

rough-state-of (*state-of* ($\llbracket \cdot \rrbracket$, N)) = ($\llbracket \cdot \rrbracket$, N)

apply (*rule* *DPLL-W-Implementation.dpll_W-state.state-of-inverse*)

unfolding *dpll_W-all-inv-def* **by** *auto*

Theorem of correctness

lemma *DPLL-tot-correct*:

assumes *rough-state-of* (*DPLL-tot* (*state-of* ($\llbracket \cdot \rrbracket$, N)))) = (M , N')

and (M' , N'') = *toS'* (M , N')

shows $M' \models_{asm} N'' \longleftrightarrow \text{satisfiable} (\text{set-mset } N'')$

proof –

have *dpll_W*** (*toS'* ($\llbracket \cdot \rrbracket$, 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-rtranclp *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

19.2.3 Code export

A conversion to *DPLL-W-Implementation*. *dpll_W-state* **definition** *Con* :: (*int*, *unit*, *unit*) *marked-lit* *list* × *int* *literal* *list* *list*

⇒ *dpll_W-state* **where**

Con *xs* = *state-of* (if *dpll_W-all-inv* (*toS* (*fst* *xs*) (*snd* *xs*)) then *xs* else ([], []))

lemma [*code abstype*]:

Con (*rough-state-of* *S*) = *S*

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* (∀ *A* ∈ *set* *N*. (∃ *a* ∈ *set* *A*. *a* ∈

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* *literal* *list*

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* *cdcl_W-marked-lits* × '*v* *literal* *list* *list* × '*v* *literal* *list* *list* × *nat* ×

'*v* *literal* *list* *option*

abbreviation *raw-trail* :: '*a* × '*b* × '*c* × '*d* × '*e* ⇒ '*a* **where**

raw-trail ≡ (λ(*M*, -). *M*)

abbreviation *raw-cons-trail* :: '*a* ⇒ '*a* *list* × '*b* × '*c* × '*d* × '*e* ⇒ '*a* *list* × '*b* × '*c* × '*d* × '*e*

where
 $raw-cons-trail \equiv (\lambda L (M, S). (L \# M, S))$

abbreviation $raw-tl-trail :: 'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$ **where**
 $raw-tl-trail \equiv (\lambda(M, S). (tl\ M, S))$

abbreviation $raw-init-clss :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b$ **where**
 $raw-init-clss \equiv \lambda(M, N, -). N$

abbreviation $raw-learned-clss :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c$ **where**
 $raw-learned-clss \equiv \lambda(M, N, U, -). U$

abbreviation $raw-backtrack-lvl :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd$ **where**
 $raw-backtrack-lvl \equiv \lambda(M, N, U, k, -). k$

abbreviation $raw-update-backtrack-lvl :: 'd \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
 $raw-update-backtrack-lvl \equiv \lambda k (M, N, U, -, S). (M, N, U, k, S)$

abbreviation $raw-conflicting :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$ **where**
 $raw-conflicting \equiv \lambda(M, N, U, k, D). D$

abbreviation $raw-update-conflicting :: 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
 $raw-update-conflicting \equiv \lambda S (M, N, U, k, -). (M, N, U, k, S)$

abbreviation $raw-add-learned-clss$ **where**
 $raw-add-learned-clss \equiv \lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

abbreviation $raw-remove-clss$ **where**
 $raw-remove-clss \equiv \lambda C (M, N, U, S). (M, removeAll-mset\ C\ N, removeAll-mset\ C\ U, S)$

type-synonym $'v\ cdcl_W\ state-inv-st = ('v, nat, 'v\ literal\ list)\ marked-lit\ list \times 'v\ literal\ list\ list \times 'v\ literal\ list\ list \times nat \times 'v\ literal\ list\ option$

abbreviation $raw-S0-cdcl_W\ N \equiv (([], N, [], 0, None) :: 'v\ cdcl_W\ state-inv-st)$

experiment
begin
interpretation $raw-clss\ mset$
 $\lambda xs\ ys. case-prod\ append\ (fold\ (\lambda x\ (ys, zs). (remove1\ x\ ys, x \# zs))\ xs\ (ys, []))$
 $op \# remove1$
by $unfold-locales\ (auto\ simp: union-mset-list\ ex-mset)$

declare $insert-clss[simp\ del]\ remove-lit[simp\ del]$
end

This is the same as $remove1$ under the assumptions of non-duplication inside a clause.

fun $remove1-eq-mset$ **where**
 $remove1-eq-mset\ -\ [] = [] \mid$
 $remove1-eq-mset\ C\ (C' \# L) = (if\ mset\ C = mset\ C'\ then\ L\ else\ C' \# remove1-eq-mset\ C\ L)$

lemma $remove1-mset-single-add$:
 $a \neq b \implies remove1-mset\ a\ (\{\#b\# \} + C) = \{\#b\# \} + remove1-mset\ a\ C$
 $remove1-mset\ a\ (\{\#a\# \} + C) = C$

```

by (auto simp: multiset-eq-iff)

lemma mset-map-mset-remove1-eq-mset:
  mset (map mset (remove1-eq-mset a C)) = remove1-mset (mset a) (mset (map mset C))
  by (induction C) (auto simp: ac-simps remove1-mset-single-add)

fun removeAll-eq-mset where
  removeAll-eq-mset - [] = [] |
  removeAll-eq-mset C (C' # L) =
    (if mset C = mset C' then removeAll-eq-mset C L else C' # removeAll-eq-mset C L)

lemma mset-map-mset-removeAll-eq-mset:
  mset (map mset (removeAll-eq-mset a C)) = removeAll-mset (mset a) (mset (map mset C))
  by (induction C) (auto simp: ac-simps mset-less-eqI multiset-diff-union-assoc)

interpretation clss-clss: raw-clss id op # $\cup$   $\lambda L$  C. C + {#L#} remove1-mset
  id op + op  $\in$   $\lambda L$  C. C + {#L#} remove1-mset
  by unfold-locales (auto simp: ac-simps)

experiment
begin
  interpretation raw-clss mset
     $\lambda xs$   $ys$ . case-prod append (fold ( $\lambda x$  (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
    op # remove1  $\lambda L$ . mset (map mset L) op @  $\lambda L$  C. L  $\in$  set C op #
    remove1-eq-mset
    by unfold-locales (auto simp: ac-simps union-mset-list mset-map-mset-remove1-eq-mset
      ex-mset)
  end

fun mmset-of-mlit' :: ('v, nat, 'v literal list) marked-lit  $\Rightarrow$  ('v, nat, 'v clause) marked-lit
  where
  mmset-of-mlit' (Propagated L C) = Propagated L (mset C) |
  mmset-of-mlit' (Marked L i) = Marked L i

lemma lit-of-mmset-of-mlit'[simp]:
  lit-of (mmset-of-mlit' xa) = lit-of xa
  by (induction xa) auto

abbreviation trail where
  trail S  $\equiv$  map mmset-of-mlit' (raw-trail S)

abbreviation clauses-of-l where
  clauses-of-l  $\equiv$   $\lambda L$ . mset (map mset L)

global-interpretation stateW-ops
  mset::'v literal list  $\Rightarrow$  'v clause
   $\lambda xs$   $ys$ . case-prod append (fold ( $\lambda x$  (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
  op # remove1

  clauses-of-l op @  $\lambda L$  C. L  $\in$  set C op # remove1-eq-mset

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

  id id

```

$\lambda(M, -). \text{map } \text{mmset-of-mlit}' M \lambda(M, -). \text{hd } M$
 $\lambda(-, N, -). N$
 $\lambda(-, -, U, -). U$
 $\lambda(-, -, -, k, -). k$
 $\lambda(-, -, -, -, C). C$

$\lambda L (M, S). (L \# M, S)$
 $\lambda(M, S). (\text{tl } M, S)$
 $\lambda C (M, N, S). (M, C \# N, S)$
 $\lambda C (M, N, U, S). (M, N, C \# U, S)$
 $\lambda C (M, N, U, S). (M, \text{removeAll-eq-mset } C N, \text{removeAll-eq-mset } C U, S)$
 $\lambda(k::\text{nat}) (M, N, U, -, D). (M, N, U, k, D)$
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$
 $\lambda N. ([], N, [], 0, \text{None})$
 $\lambda(-, N, U, -). ([], N, U, 0, \text{None})$
apply *unfold-locales* **by** (*auto simp: hd-map comp-def map-tl ac-simps*
union-mset-list mset-map-mset-remove1-eq-mset ex-mset)

lemma *mmset-of-mlit'-mmset-of-mlit*: *mmset-of-mlit' l = mmset-of-mlit l*
apply (*induct l*)
apply *auto*
done

interpretation *statew*
mset::'v literal list \Rightarrow 'v clause
 $\lambda xs \text{ ys. case-prod append (fold } (\lambda x (ys, zs). (\text{remove1 } x \text{ ys}, x \# zs)) xs (ys, []))$
 $op \# \text{remove1}$

$\lambda L. \text{mset } (\text{map } \text{mset } L) \text{ op } @ \lambda L C. L \in \text{set } C \text{ op } \# \text{remove1-eq-mset}$

$\text{mset } \lambda xs \text{ ys. case-prod append (fold } (\lambda x (ys, zs). (\text{remove1 } x \text{ ys}, x \# zs)) xs (ys, []))$
 $op \# \text{remove1}$

id id

$\lambda(M, -). \text{map } \text{mmset-of-mlit}' M \lambda(M, -). \text{hd } M$
 $\lambda(-, N, -). N$
 $\lambda(-, -, U, -). U$
 $\lambda(-, -, -, k, -). k$
 $\lambda(-, -, -, -, C). C$

$\lambda L (M, S). (L \# M, S)$
 $\lambda(M, S). (\text{tl } M, S)$
 $\lambda C (M, N, S). (M, C \# N, S)$
 $\lambda C (M, N, U, S). (M, N, C \# U, S)$
 $\lambda C (M, N, U, S). (M, \text{removeAll-eq-mset } C N, \text{removeAll-eq-mset } C U, S)$
 $\lambda(k::\text{nat}) (M, N, U, -, D). (M, N, U, k, D)$
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$
 $\lambda N. ([], N, [], 0, \text{None})$
 $\lambda(-, N, U, -). ([], N, U, 0, \text{None})$
apply *unfold-locales*
apply (*rename-tac S, case-tac S*)
by (*auto simp: hd-map comp-def map-tl ac-simps mset-map-mset-removeAll-eq-mset*
mmset-of-mlit'-mmset-of-mlit)

global-interpretation *conflict-driven-clause-learning_W*

mset::'v literal list \Rightarrow 'v clause

$\lambda xs\ ys. \text{case-prod append (fold (\lambda x (ys, zs). (remove1 x ys, x \# zs)) xs (ys, []))$

op # remove1

$\lambda L. \text{mset (map mset L) op @ } \lambda L\ C. L \in \text{set } C\ \text{op \# remove1-eq-mset}$

$\text{mset } \lambda xs\ ys. \text{case-prod append (fold (\lambda x (ys, zs). (remove1 x ys, x \# zs)) xs (ys, []))$

op # remove1

id id

$\lambda(M, -). \text{map mmset-of-mlit'}\ M\ \lambda(M, -). \text{hd } M$

$\lambda(-, N, -). N$

$\lambda(-, -, U, -). U$

$\lambda(-, -, -, k, -). k$

$\lambda(-, -, -, -, C). C$

$\lambda L\ (M, S). (L \# M, S)$

$\lambda(M, S). (\text{tl } M, S)$

$\lambda C\ (M, N, S). (M, C \# N, S)$

$\lambda C\ (M, N, U, S). (M, N, C \# U, S)$

$\lambda C\ (M, N, U, S). (M, \text{removeAll-eq-mset } C\ N, \text{removeAll-eq-mset } C\ U, S)$

$\lambda(k::\text{nat})\ (M, N, U, -, D). (M, N, U, k, D)$

$\lambda D\ (M, N, U, k, -). (M, N, U, k, D)$

$\lambda N. ([], N, [], 0, \text{None})$

$\lambda(-, N, U, -). ([], N, U, 0, \text{None})$

by *intro-locales*

declare *state-simp[simp del] raw-clauses-def[simp] state-eq-def[simp]*

notation *state-eq (infix \sim 50)*

term *reduce-trail-to*

lemma *reduce-trail-to-map[simp]:*

reduce-trail-to (map f M1) = reduce-trail-to M1

by *(rule ext) (auto intro: reduce-trail-to-length)*

19.3 CDCL Implementation

19.3.1 Definition of the rules

Types **lemma** *true-clss-remdups[simp]:*

$I \models_s (\text{mset} \circ \text{remdups})\ 'N \longleftrightarrow I \models_s \text{mset}\ 'N$

by *(simp add: true-clss-def)*

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

satisfiable ((mset \circ remdups) 'N) \longleftrightarrow satisfiable (mset 'N)

unfolding *satisfiable-carac[symmetric] by simp*

We need some functions to convert between our abstract state *nat cdcl_W-state* and the concrete state *'v cdcl_W-state-inv-st*.

abbreviation *convertC :: 'a list option \Rightarrow 'a multiset option* **where**

convertC \equiv map-option mset

lemma *convert-Propagated[elim!]:*

mmset-of-mlit' $z = \text{Propagated } L \ C \implies (\exists C'. z = \text{Propagated } L \ C' \wedge C = \text{mset } C')$
by (*cases* z) *auto*

lemma *get-rev-level-map-convert*:

get-rev-level (*map* *mmset-of-mlit'* M) $n \ x = \text{get-rev-level } M \ n \ x$
by (*induction* M *arbitrary*; n *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-map-convert*[*simp*]:

get-level (*map* *mmset-of-mlit'* M) = *get-level* M
using *get-rev-level-map-convert*[*of* $\text{rev } M$] **by** (*simp* *add*: *rev-map*)

lemma *get-rev-level-map-mmsetof-mlit*[*simp*]:

get-rev-level (*map* *mmset-of-mlit* M) = *get-rev-level* M
by (*induction* M *rule*: *marked-lit-list-induct*) (*auto* *intro!*: *ext*)

lemma *get-level-map-mmsetof-mlit*[*simp*]:

get-level (*map* *mmset-of-mlit* M) = *get-level* M
using *get-rev-level-map-mmsetof-mlit*[*of* $\text{rev } M$] **unfolding** *rev-map* **by** *simp*

lemma *get-maximum-level-map-convert*[*simp*]:

get-maximum-level (*map* *mmset-of-mlit'* M) $D = \text{get-maximum-level } M \ D$
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* *mmset-of-mlit'* M) = (*get-all-levels-of-marked* M)
by (*induction* M *rule*: *marked-lit-list-induct*) *auto*

lemma *reduce-trail-to-empty-trail*[*simp*]:

reduce-trail-to $F \ (\[], aa, ab, ac, b) = (\[], aa, ab, ac, b)$
using *reduce-trail-to.simps* **by** *auto*

lemma *raw-trail-reduce-trail-to-length-le*:

assumes *length* $F > \text{length} \ (\text{raw-trail } S)$
shows *raw-trail* (*reduce-trail-to* $F \ S$) = $\[]$
using *assms* *trail-reduce-trail-to-length-le*[*of* $S \ F$]
by (*cases* S , *cases* *reduce-trail-to* $F \ S$) *auto*

lemma *reduce-trail-to*:

reduce-trail-to $F \ S =$
 ((*if* *length* (*raw-trail* S) $\geq \text{length } F$
 then *drop* (*length* (*raw-trail* S) $- \text{length } F$) (*raw-trail* S)
 else $\[]$), *raw-init-clss* S , *raw-learned-clss* S , *raw-backtrack-lvl* S , *raw-conflicting* S)
 (*is* $?S = -$)

proof (*induction* $F \ S$ *rule*: *reduce-trail-to.induct*)

case ($1 \ F \ S$) **note** $IH = \text{this}$

show $?case$

proof (*cases* *raw-trail* S)

case *Nil*

then show $?thesis$ **using** IH **by** (*cases* S) *auto*

next

case (*Cons* $L \ M$)

then show $?thesis$

apply (*cases* *Suc* (*length* M) $> \text{length } F$)

prefer 2 **using** IH *reduce-trail-to-length-ne*[*of* $S \ F$] **apply** (*cases* S) **apply** *auto*

apply (*subgoal-tac* *Suc* (*length* M) $- \text{length } F = \text{Suc} \ (\text{length } M - \text{length } F)$)

```

    using reduce-trail-to-length-ne[of S F] IH by (cases S) (auto simp add:)
  qed
qed

```

Definition an abstract type

```

typedef 'v cdclW-state-inv = {S::'v cdclW-state-inv-st. cdclW-all-struct-inv S}
morphisms rough-state-of state-of
proof
  show ([], [], [], 0, None) ∈ {S. cdclW-all-struct-inv S}
  by (auto simp add: cdclW-all-struct-inv-def)
qed

instantiation cdclW-state-inv :: (type) equal
begin
definition equal-cdclW-state-inv :: 'v cdclW-state-inv ⇒ 'v cdclW-state-inv ⇒ bool where
  equal-cdclW-state-inv S S' = (rough-state-of S = rough-state-of S')
instance
  by standard (simp add: rough-state-of-inject equal-cdclW-state-inv-def)
end

```

```

lemma lits-of-map-convert[simp]: lits-of-l (map mmset-of-mlit' M) = lits-of-l M
by (induction M rule: marked-lit-list-induct) simp-all

```

```

lemma undefined-lit-map-convert[iff]:
  undefined-lit (map mmset-of-mlit' M) L ⟷ undefined-lit M L
by (auto simp add: defined-lit-map image-image mmset-of-mlit'-mmset-of-mlit)

```

```

lemma true-annot-map-convert[simp]: map mmset-of-mlit' M ⊨a N ⟷ M ⊨a N
by (induction M rule: marked-lit-list-induct) (simp-all add: true-annot-def
  mmset-of-mlit'-mmset-of-mlit)

```

```

lemma true-annots-map-convert[simp]: map mmset-of-mlit' M ⊨as N ⟷ M ⊨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 (M, N, U, k, None) (Propagated L C # M, N, U, k, None)
  using assms
  by (auto dest!: find-first-unit-clause-some intro!: propagate-rule)

```

19.3.2 The Transitions

Propagate **definition** do-propagate-step **where**

```

do-propagate-step S =
  (case S of
    (M, N, U, k, None) ⇒
      (case find-first-unit-clause (N @ U) M of
        Some (L, C) ⇒ (Propagated L C # M, N, U, k, None)
      | None ⇒ (M, N, U, k, None))
  | S ⇒ S)

```

```

lemma do-propagate-step:
  do-propagate-step S ≠ S ⟹ propagate S (do-propagate-step S)
apply (cases S, cases conflicting S)
using find-first-unit-clause-some-is-propagate[of raw-init-clss S raw-learned-clss S]

```

```

by (auto simp add: do-propagate-step-def split: option.splits)

lemma do-propagate-step-option[simp]:
  conflicting  $S \neq \text{None} \implies \text{do-propagate-step } S = S$ 
  unfolding do-propagate-step-def by (cases  $S$ , cases conflicting  $S$ ) auto
thm prod-cases

lemma do-propagate-step-no-step:
  assumes  $\text{dist}: \forall c \in \text{set } (\text{raw-clauses } S). \text{distinct } c$  and
  prop-step:  $\text{do-propagate-step } S = S$ 
  shows no-step propagate  $S$ 
proof (standard, standard)
  fix  $T$ 
  assume propagate  $S T$ 
  then obtain  $C L$  where
    toSS: conflicting  $S = \text{None}$  and
     $C: C \in \text{set } (\text{raw-clauses } S)$  and
     $L: L \in \text{set } C$  and
     $MC: \text{raw-trail } S \models_{\text{as}} C \text{Not } (\text{mset } (\text{remove1 } L C))$  and
     $T: T \sim \text{raw-cons-trail } (\text{Propagated } L C) S$  and
    undef: undefined-lit  $(\text{raw-trail } S) L$ 
  apply (cases  $S$  rule: prod-cases5)
  by (elim propagateE) simp
  let ? $M$  = raw-trail  $S$ 
  let ? $N$  = raw-init-clss  $S$ 
  let ? $U$  = raw-learned-clss  $S$ 
  let ? $k$  = raw-backtrack-lvl  $S$ 
  let ? $D$  = None
  have  $S: S = (?M, ?N, ?U, ?k, ?D)$ 
    using toSS by (cases  $S$ , cases conflicting  $S$ ) simp-all

  have find-first-unit-clause  $(?N @ ?U) ?M \neq \text{None}$ 
  apply (rule dist find-first-unit-clause-none[of  $C ?N @ ?U ?M L$ , OF -])
  using  $C$  dist apply auto[]
  using  $C$  apply auto[1]
  using  $MC$  apply auto[1]
  using undef apply auto[1]
  using  $L$  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  $M [] = \text{None} \mid$ 
  find-conflict  $M (N \# Ns) = (\text{if } (\forall c \in \text{set } N. -c \in \text{lits-of-l } M) \text{ then Some } N \text{ else find-conflict } M Ns)$ 

lemma find-conflict-Some:
  find-conflict  $M Ns = \text{Some } N \implies N \in \text{set } Ns \wedge M \models_{\text{as}} C \text{Not } (\text{mset } N)$ 
  by (induction  $Ns$  rule: find-conflict.induct)
  (auto split: if-split-asm)

lemma find-conflict-None:
  find-conflict  $M Ns = \text{None} \longleftrightarrow (\forall N \in \text{set } Ns. \neg M \models_{\text{as}} C \text{Not } (\text{mset } N))$ 
  by (induction  $Ns$ ) auto

lemma find-conflict-None-no-confl:

```

find-conflict $M (N @ U) = \text{None} \longleftrightarrow \text{no-step conflict } (M, N, U, k, \text{None})$
by (auto simp add: find-conflict-None conflict.simps)

definition *do-conflict-step* **where**

do-conflict-step $S =$
 (case S of
 $(M, N, U, k, \text{None}) \Rightarrow$
 (case *find-conflict* $M (N @ U)$ of
 Some $a \Rightarrow (M, N, U, k, \text{Some } a)$
 None $\Rightarrow (M, N, U, k, \text{None})$)
 | $S \Rightarrow S$)

lemma *do-conflict-step*:

do-conflict-step $S \neq S \implies \text{conflict } S$ (*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 simp: state-eq-def)

lemma *do-conflict-step-no-step*:

do-conflict-step $S = S \implies \text{no-step conflict } S$
apply (cases S , cases conflicting S)
unfolding *do-conflict-step-def*
using find-conflict-None-no-conf[*of* raw-trail S raw-init-clss S raw-learned-clss S
 raw-backtrack-lvl S]
by (auto split: option.split elim: conflictE)

lemma *do-conflict-step-option[simp]*:

conflicting $S \neq \text{None} \implies \text{do-conflict-step } S = S$
unfolding *do-conflict-step-def* **by** (cases S , cases conflicting S) auto

lemma *do-conflict-step-conflicting[dest]*:

do-conflict-step $S \neq S \implies \text{conflicting } (\text{do-conflict-step } S) \neq \text{None}$
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-cdcl_W-cp*:

assumes H : *do-cp-step* $S \neq S$
shows *cdcl_W-cp* S (*do-cp-step* S)

proof –

show ?thesis

proof (cases *do-conflict-step* $S \neq S$)

case True

then have *do-propagate-step* (*do-conflict-step* S) = *do-conflict-step* S

by auto

then show ?thesis

by (auto simp add: *do-conflict-step* *do-conflict-step-conflicting* *do-cp-step-def* True)

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 = S
let ?T = (do-propagate-step S)
let ?U = (do-conflict-step (do-propagate-step S))
have propa: propagate S ?T using False do-propagate-step by blast
moreover have ns: no-step conflict 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

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 raw-conflicting S)
  (auto simp add: do-conflict-step-def do-propagate-step-def do-cp-step-def split: option.splits)

lemma no-cdclW-cp-iff-no-propagate-no-conflict:
  no-step cdclW-cp S  $\longleftrightarrow$  no-step propagate S  $\wedge$  no-step conflict S
by (auto simp: cdclW-cp.simps)

lemma do-cp-step-eq-no-step:
  assumes
    H: do-cp-step S = S and
     $\forall c \in \text{set } (\text{raw-init-clss } S @ \text{raw-learned-clss } S). \text{ distinct } c$ 
  shows no-step cdclW-cp S
  unfolding no-cdclW-cp-iff-no-propagate-no-conflict
  using assms apply (cases S, cases conflicting S)
  using do-propagate-step-no-step[of S]
  by (auto dest!: do-cp-step-eq-no-prop-no-confl[simplified] do-conflict-step-no-step
    split: option.splits)

lemma cdclW-cp-cdclW-st: cdclW-cp S S'  $\implies$  cdclW** S S'
by (simp add: cdclW-cp-tranclp-cdclW tranclp-into-rtranclp)

lemma cdclW-all-struct-inv-rough-state[simp]: cdclW-all-struct-inv (rough-state-of S)
  using rough-state-of by auto

lemma [simp]: cdclW-all-struct-inv 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 cdclW-all-struct-inv (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-cdclW-cp[of rough-state-of S] cdclW-all-struct-inv-rough-state[of S]
  cdclW-cp-cdclW-st rtranclp-cdclW-all-struct-inv-inv by blast
  then show ?thesis by auto
qed

```

Skip **fun** *do-skip-step* :: 'v *cdcl_W-state-inv-st* \Rightarrow 'v *cdcl_W-state-inv-st* **where**
do-skip-step (*Propagated* *L C # Ls, N, U, k, Some D*) =
 (if $\neg L \in \text{set } D \wedge D \neq []$
 then (*Ls, N, U, k, Some D*)
 else (*Propagated L C # Ls, N, U, k, Some D*)) |
do-skip-step S = S

lemma *do-skip-step*:
do-skip-step S \neq S \implies skip S (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 no-step skip S
by (*induction S rule: do-skip-step.induct*)
 (*auto simp add: other split: if-split-asm elim!: skipE*)

lemma *do-skip-step-trail-is-None[iff]*:
do-skip-step S = (a, b, c, d, None) \longleftrightarrow S = (a, b, c, d, None)
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 (get-level M L) (maximum-level-code Ls M)

lemma *maximum-level-code-eq-get-maximum-level[code, simp]*:
maximum-level-code D M = get-maximum-level M (mset D)
by (*induction D*) (*auto simp add: get-maximum-level-plus*)

fun *do-resolve-step* :: 'v *cdcl_W-state-inv-st* \Rightarrow 'v *cdcl_W-state-inv-st* **where**
do-resolve-step (*Propagated L C # Ls, N, U, k, Some D*) =
 (if $\neg L \in \text{set } D \wedge \text{maximum-level-code (remove1 } (\neg L) D) (\text{Propagated } L C \# Ls) = k$
 then (*Ls, N, U, k, Some (remdups (remove1 L C @ remove1 } (\neg L) D))*)
 else (*Propagated L C # Ls, N, U, k, Some D*)) |
do-resolve-step S = S

lemma *do-resolve-step*:
cdcl_W-all-struct-inv S \implies do-resolve-step S \neq S
 \implies *resolve S (do-resolve-step S)*

proof (*induction S rule: do-resolve-step.induct*)

case (*1 L C M N U k D*)

then have

LD: $\neg L \in \text{set } D$ and

M: maximum-level-code (remove1 } (\neg L) D) (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: if-split-asm)+

have *every-mark-is-a-conflict (Propagated L C # M, N, U, k, Some D)*

using *1(1) unfolding cdcl_W-all-struct-inv-def cdcl_W-conflicting-def* **by** *fast*

then have *LC: L \in 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 *($\neg L \in \text{set } D$)* **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 max: get-maximum-level (Propagated L (C' + {#L#}) # map mmset-of-mlit' M) D' = k
  using M[simplified] unfolding maximum-level-code-eq-get-maximum-level C[symmetric] CL
  by (metis D D'L get-maximum-level-map-convert list.simps(9) mmset-of-mlit'.simps(1))
have distinct-mset (mset C) and distinct-mset (mset D)
  using ⟨cdclW-all-struct-inv (Propagated L C # M, N, U, k, Some D)⟩
  unfolding cdclW-all-struct-inv-def distinct-cdclW-state-def
  by auto
then have conf: (mset C - {#L#}) # ∪ (mset D - {#- L#}) =
  remdups-mset (mset C - {#L#} + (mset D - {#- L#}))
  by (auto simp: distinct-mset-remdups-union-mset)
show ?case
  apply (rule resolve-rule)
  using LC LD max M conf C D by (auto simp: subset-mset.sup commute)
qed auto

```

```

lemma do-resolve-step-no:
  do-resolve-step S = S ⟹ no-step resolve S
  apply (cases S; cases (raw-trail S); cases raw-conflicting S)
  by (auto
      elim!: resolveE split: if-split-asm
      dest!: union-single-eq-member
      simp del: in-multiset-in-set get-maximum-level-map-convert
      simp: get-maximum-level-map-convert[symmetric] do-resolve-step)

```

```

lemma rough-state-of-state-of-resolve[simp]:
  cdclW-all-struct-inv S ⟹ rough-state-of (state-of (do-resolve-step S)) = do-resolve-step S
  apply (rule state-of-inverse)
  apply (cases do-resolve-step S = S)
  apply simp
  by (blast dest: other resolve bj do-resolve-step cdclW-all-struct-inv-inv)

```

```

lemma do-resolve-step-trail-is-None[iff]:
  do-resolve-step S = (a, b, c, d, None) ⟷ S = (a, b, c, d, None)
  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 M L, maximum-level-code (D @ Ls) M) of
      (i, j) ⇒ if i = k ∧ j < i then Some (L, j) else find-level-decomp M Ls (L#D) k
    )

```

```

lemma find-level-decomp-some:
  assumes find-level-decomp M Ls D k = Some (L, j)
  shows L ∈ set Ls ∧ get-maximum-level M (mset (remove1 L (Ls @ D))) = j ∧ get-level M L = k
  using assms
proof (induction Ls arbitrary: D)
  case Nil
  then show ?case by simp
next
  case (Cons L' Ls) note IH = this(1) and H = this(2)

```

```

def find ≡ (if get-level M L' ≠ k ∨ ¬ get-maximum-level M (mset D + mset Ls) < get-level M L'
  then find-level-decomp M Ls (L' # D) k
  else Some (L', get-maximum-level M (mset D + mset Ls)))
have a1: ∧ D. find-level-decomp M Ls D k = Some (L, j) ⇒
  L ∈ set Ls ∧ get-maximum-level M (mset Ls + mset D - {#L#}) = j ∧ get-level M L = k
using IH by simp
have a2: find = Some (L, j)
  using H unfolding find-def by (auto split: if-split-asm)
{ assume Some (L', get-maximum-level M (mset D + mset Ls)) ≠ find
  then have f3: L ∈ set Ls and get-maximum-level M (mset Ls + mset (L' # D) - {#L#}) = j
    using a1 IH a2 unfolding find-def by meson+
    moreover then have mset Ls + mset D - {#L#} + {#L'#} = {#L'#} + mset D + (mset Ls
- {#L#})
    by (auto simp: ac-simps multiset-eq-iff Suc-leI)
    ultimately have f4: get-maximum-level M (mset Ls + mset D - {#L#} + {#L'#}) = j
    by (metis add.commute diff-union-single-conv in-multiset-in-set mset.simps(2))
  } note f4 = this
have {#L'#} + (mset Ls + mset D) = mset Ls + (mset D + {#L'#})
  by (auto simp: ac-simps)
then have
  (L = L' → get-maximum-level M (mset Ls + mset D) = j ∧ get-level M L' = k) and
  (L ≠ L' → L ∈ set Ls ∧ get-maximum-level M (mset Ls + mset D - {#L#} + {#L'#}) = j ∧
  get-level M L = k)
  using f4 a2 a1 [of L' # D] unfolding find-def by (metis (no-types) add-diff-cancel-left'
  mset.simps(2) option.inject prod.inject union-commute)+
then show ?case by simp
qed

```

lemma find-level-decomp-none:

```

assumes find-level-decomp M Ls E k = None and mset (L # D) = mset (Ls @ E)
shows ¬(L ∈ set Ls ∧ get-maximum-level M (mset D) < k ∧ k = get-level M L)
using assms
proof (induction Ls arbitrary: E L D)
  case Nil
  then show ?case by simp
next
  case (Cons L' Ls) note IH = this(1) and find-none = this(2) and LD = this(3)
  have mset D + {#L'#} = mset E + (mset Ls + {#L'#}) ⇒ mset D = mset E + mset Ls
    by (metis add-right-imp-eq union-assoc)
  then show ?case
    using find-none IH [of L' # E L D] LD by (auto simp add: ac-simps split: if-split-asm)
qed

```

fun bt-cut where

```

bt-cut i (Propagated - - # Ls) = bt-cut i Ls |
bt-cut i (Marked K k # Ls) = (if k = Suc i then Some (Marked K k # Ls) else bt-cut i Ls) |
bt-cut i [] = None

```

lemma bt-cut-some-decomp:

```

bt-cut i M = Some M' ⇒ ∃ K M2 M1. M = M2 @ M' ∧ M' = Marked K (i+1) # M1
by (induction i M rule: bt-cut.induct) (auto split: if-split-asm)

```

lemma bt-cut-not-none: $M = M2 @ \text{Marked } K (\text{Suc } i) \# M' \Rightarrow \text{bt-cut } i M \neq \text{None}$

```

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 (*rename-tac* *L m xs*, *case-tac* *get-all-marked-decomposition* (*xs @ Marked K (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, Some D*) =

(*case find-level-decomp M D [] k of*

None \Rightarrow (*M, N, U, k, Some D*)

| *Some (L, j)* \Rightarrow

(*case bt-cut j M of*

Some (Marked - - # Ls) \Rightarrow (*Propagated L D # Ls, N, D # U, j, None*)

| - \Rightarrow (*M, N, U, k, Some D*))

) |

do-backtrack-step S = S

lemma *get-all-marked-decomposition-map-convert*:

(*get-all-marked-decomposition* (*map mmset-of-mlit' M*)) =

map ($\lambda(a, b). (\text{map mmset-of-mlit' } a, \text{map mmset-of-mlit' } b)$) (*get-all-marked-decomposition M*)

apply (*induction M* *rule*: *marked-lit-list-induct*)

apply *simp*

by (*rename-tac L l xs*, *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 S*

shows *backtrack S (do-backtrack-step S)*

proof (*cases S*, *cases raw-conflicting S*, *goal-cases*)

case (*1 M N U k E*)

then show *?case* **using** *db* **by** *auto*

next

case (*2 M N U k E C*) **note** *S = this(1)* **and** *confl = this(2)*

have *E*: *E = Some C* **using** *S confl* **by** *auto*

obtain *L j* **where** *fd*: *find-level-decomp M C [] k = Some (L, j)*

using *db unfolding S E* **by** (*cases C*) (*auto split: if-split-asm option.splits*)

have

L \in *set C* **and**

j: *get-maximum-level M (mset (remove1 L C)) = j* **and**

levL: *get-level M L = k*

using *find-level-decomp-some[OF fd]* **by** *auto*

obtain *C'* **where** *C*: *mset C = mset C' + {#L#}*

using (*L* \in *set C*) **by** (*metis add.commute ex-mset in-multiset-in-set insert-DiffM*)

obtain *M₂* **where** *M₂*: *bt-cut j M = Some M₂*

using *db fd unfolding S E* **by** (*auto split: option.splits*)

obtain *M1 K* **where** *M1*: *M₂ = Marked K (Suc j) # M1*

using *bt-cut-some-decomp[OF M₂]* **by** (*cases M₂*) *auto*

```

obtain  $c$  where  $c: M = c @ \text{Marked } K (\text{Suc } j) \# M1$ 
  using  $\text{bt-cut-in-get-all-marked-decomposition}[OF M_2]$ 
  unfolding  $M1$  by  $\text{fastforce}$ 
have  $\text{get-all-levels-of-marked } (\text{map } \text{mmset-of-mlit}' M) = \text{rev } [1..<\text{Suc } k]$ 
  using  $\text{inv unfolding cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def } S$  by  $\text{auto}$ 
from  $\text{arg-cong}[OF \text{this}, \text{of } \lambda a. \text{Suc } j \in \text{set } a]$  have  $j \leq k$  unfolding  $c$  by  $\text{auto}$ 
have  $\text{max-l-j: maximum-level-code } C' M = j$ 
  using  $\text{db fd } M_2 C$  unfolding  $S E$  by  $(\text{auto}$ 
     $\text{split: option.splits list.splits marked-lit.splits}$ 
     $\text{dest! : find-level-decomp-some})[1]$ 
have  $\text{get-maximum-level } M (\text{mset } C) \geq k$ 
  using  $\langle L \in \text{set } C \rangle \text{ levL get-maximum-level-ge-get-level }$  by  $(\text{metis set-mset-mset})$ 
moreover have  $\text{get-maximum-level } M (\text{mset } C) \leq k$ 
  using  $\text{get-maximum-level-exists-lit-of-max-level}[of \text{mset } C M]$   $\text{inv}$ 
   $\text{cdcl}_W\text{-M-level-inv-get-level-le-backtrack-lvl}[of S]$ 
  unfolding  $C \text{ cdcl}_W\text{-all-struct-inv-def } S$  by  $(\text{auto dest: sym}[of \text{get-level - } -])$ 
ultimately have  $\text{get-maximum-level } M (\text{mset } C) = k$  by  $\text{auto}$ 

obtain  $M2$  where  $M2: (M_2, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
  using  $\text{bt-cut-in-get-all-marked-decomposition}[OF M_2]$  by  $\text{metis}$ 
have  $\text{decomp:}$ 
   $(\text{Marked } K (\text{Suc } (\text{get-maximum-level } M (\text{remove1-mset } L (\text{mset } C)))) \# (\text{map } \text{mmset-of-mlit}' M1),$ 
   $(\text{map } \text{mmset-of-mlit}' M2)) \in$ 
   $\text{set } (\text{get-all-marked-decomposition } (\text{map } \text{mmset-of-mlit}' M))$ 
  using  $\text{imageI}[of - - \lambda(a, b). (\text{map } \text{mmset-of-mlit}' a, \text{map } \text{mmset-of-mlit}' b), OF M2]$   $j$ 
  unfolding  $S E M1$  by  $(\text{auto simp add: get-all-marked-decomposition-map-convert})$ 
have  $\text{red: } (\text{reduce-trail-to } (\text{map } \text{mmset-of-mlit}' M1)$ 
   $(M, N, C \# U, \text{get-maximum-level } M (\text{remove1-mset } L (\text{mset } C)), \text{None}))$ 
   $= (M1, N, C \# U, \text{get-maximum-level } M (\text{remove1-mset } L (\text{mset } C)), \text{None}))$ 
  using  $M2 M1$  by  $(\text{auto simp: reduce-trail-to})$ 
show  $?case$ 
  apply  $(\text{rule backtrack-rule})$ 
  using  $M_2 \text{ fd confl } \langle L \in \text{set } C \rangle j \text{ decomp levL } \langle \text{get-maximum-level } M (\text{mset } C) = k \rangle$ 
  unfolding  $S E M1$  apply  $(\text{auto simp: mset-map})[6]$ 
  unfolding  $\text{CDCL-W-Implementation.state-eq-def}$ 
  using  $M_2 \text{ fd confl } \langle L \in \text{set } C \rangle j \text{ decomp levL } \langle \text{get-maximum-level } M (\text{mset } C) = k \rangle \text{ red}$ 
  unfolding  $S E M1$ 
  by  $\text{auto}$ 

```

qed

lemma $\text{map-eq-list-length:}$

$\text{map } f L = L' \implies \text{length } L = \text{length } L'$

by auto

lemma $\text{map-mmset-of-mlit-eq-cons:}$

assumes $\text{map } \text{mmset-of-mlit}' M = a @ c$

obtains $a' c'$ **where**

$M = a' @ c'$ **and**

$a = \text{map } \text{mmset-of-mlit}' a'$ **and**

$c = \text{map } \text{mmset-of-mlit}' c'$

using $\text{that}[of \text{take } (\text{length } a) M \text{ drop } (\text{length } a) M]$

assms **by** $(\text{metis append-eq-conv-conj append-take-drop-id drop-map take-map})$

lemma $\text{do-backtrack-step-no:}$

assumes

db: *do-backtrack-step* $S = S$ **and**
inv: *cdcl_W-all-struct-inv* S
shows *no-step backtrack* S
proof (*rule ccontr*, *cases S*, *cases conflicting S*, *goal-cases*)
case 1
then show ?*case* **using** *db* **by** (*auto split: option.splits elim: backtrackE*)
next
case (2 $M N U k E C$) **note** $bt = this(1)$ **and** $S = this(2)$ **and** $confl = this(3)$
obtain $K j M1 M2 L D$ **where**
CE: *raw-conflicting* $S = Some D$ **and**
LD: $L \in \# mset D$ **and**
decomp: (*Marked* $K (Suc j) \# M1, M2$) $\in set (get-all-marked-decomposition (trail S))$ **and**
levL: *get-level* (*raw-trail* S) $L = raw-backtrack-lvl S$ **and**
k: *get-level* (*raw-trail* S) $L = get-maximum-level (raw-trail S) (mset D)$ **and**
j: *get-maximum-level* (*raw-trail* S) (*remove1-mset* $L (mset D)$) $\equiv j$ **and**
undef: *undefined-lit* $M1 L$
using bt **apply** *clarsimp*
apply (*elim backtrack-levE*)
using *inv unfolding cdcl_W-all-struct-inv-def* **apply** *fast*
apply (*cases S*)
by (*auto simp add: get-all-marked-decomposition-map-convert*)

obtain c **where** $c: trail S = c @ M2 @ Marked K (Suc j) \# M1$
using *decomp* **by** *blast*
have *get-all-levels-of-marked* (*trail* S) $= rev [1..<Suc k]$
using *inv unfolding cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def* S **by** *auto*
from *arg-cong[OF this, of $\lambda a. Suc j \in set a$]* **have** $k > j$
unfolding c **by** (*auto simp: get-all-marked-decomposition-map-convert*)
have [*simp*]: $L \in set D$
using *LD* **by** *auto*
have $CD: C = mset D$
using *CE confl* **by** *auto*
obtain D' **where**
E: $E = Some D$ **and**
DD': $mset D = \{\#L\# \} + mset D'$
using *that[of remove1 L D]*
using $S CE confl LD$ **by** (*auto simp add: insert-DiffM*)
have *find-level-decomp* $M D$ $\sqcap k \neq None$
apply *rule*
apply (*drule find-level-decomp-none[of - - - L D']*)
using $DD' \langle k > j \rangle mset-eq-setD S levL$ **unfolding** $k[symmetric] j[symmetric]$
by (*auto simp: ac-simps*)
then obtain $L' j'$ **where** *fd-some*: *find-level-decomp* $M D$ $\sqcap k = Some (L', j')$
by (*cases find-level-decomp M D* $\sqcap k$) *auto*
have $L': L' = L$
proof (*rule ccontr*)
assume $\neg ?thesis$
then have $L' \in \# mset (remove1 L D)$
by (*metis fd-some find-level-decomp-some in-set-remove1 set-mset-mset*)
then have *get-level* $M L' \leq get-maximum-level M (mset (remove1 L D))$
using *get-maximum-level-ge-get-level* **by** *blast*
then show *False* **using** $\langle k > j \rangle j$ *find-level-decomp-some[OF fd-some]* $S DD'$ **by** *auto*
qed
then have $j': j' = j$ **using** *find-level-decomp-some[OF fd-some]* $j S DD'$ **by** *auto*

```

obtain  $c' M1'$  where  $cM: M = c' @ \text{Marked } K (Suc\ j) \# M1'$ 
apply (rule map-mmset-of-mlit-eq-cons[of  $M\ c\ @\ M2\ \text{Marked } K (Suc\ j) \# M1'$ ])
using  $c\ S$  apply simp
apply (rule map-mmset-of-mlit-eq-cons[of - [Marked  $K (Suc\ j)$ ]  $M1'$ ])
apply auto[]
apply (rename-tac  $a\ b'$   $aa\ b$ , case-tac  $aa$ )
apply auto[]
apply (rename-tac  $a\ b'$   $aa\ b$ , case-tac  $aa$ )
by auto
have btc-none: bt-cut  $j\ M \neq \text{None}$ 
apply (rule bt-cut-not-none[of  $M$  ])
using  $cM$  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  $S$ 
  shows rough-state-of (state-of (do-backtrack-step  $S$ )) = do-backtrack-step  $S$ 
proof (rule state-of-inverse)
  have f2: backtrack  $S$  (do-backtrack-step  $S$ )  $\vee$  do-backtrack-step  $S = S$ 
    using do-backtrack-step inv by blast
  have  $\bigwedge p. \neg \text{cdcl}_W\text{-o } S\ p \vee \text{cdcl}_W\text{-all-struct-inv } p$ 
    using inv cdclW-all-struct-inv-inv other by blast
  then have do-backtrack-step  $S = S \vee \text{cdcl}_W\text{-all-struct-inv (do-backtrack-step } S)$ 
    using f2 inv cdclW-o.intros cdclW-bj.intros by blast
  then show do-backtrack-step  $S \in \{S. \text{cdcl}_W\text{-all-struct-inv } S\}$ 
    using inv by fastforce
qed

```

```

Decide fun do-decide-step where
  do-decide-step ( $M, N, U, k, \text{None}$ ) =
    (case find-first-unused-var  $N$  (lits-of-l  $M$ ) of
       $\text{None} \Rightarrow (M, N, U, k, \text{None})$ 
    |  $\text{Some } L \Rightarrow (\text{Marked } L (Suc\ k) \# M, N, U, k+1, \text{None}))$  |
  do-decide-step  $S = S$ 

```

```

lemma do-decide-step:
  fixes  $S :: 'v\ \text{cdcl}_W\text{-state-inv-st}$ 
  assumes do-decide-step  $S \neq S$ 
  shows decide  $S$  (do-decide-step  $S$ )
  using assms
  apply (cases  $S$ , cases conflicting  $S$ )
  defer
  apply (auto split: option.splits simp add: decide.simps Marked-Propagated-in-iff-in-lits-of-l
    dest: find-first-unused-var-undefined find-first-unused-var-Some
    intro:)[1]
proof -
  fix  $a :: ('v, \text{nat}, 'v\ \text{literal list})\ \text{marked-lit list}$  and
     $b :: 'v\ \text{literal list list}$  and  $c :: 'v\ \text{literal list list}$  and
     $d :: \text{nat}$  and  $e :: 'v\ \text{literal list option}$ 
  {
    fix  $a :: ('v, \text{nat}, 'v\ \text{literal list})\ \text{marked-lit list}$  and

```



```

    b :: 'v literal list list and c :: 'v literal list list and
    d :: nat and x2 :: 'v literal and m :: 'v literal list
  assume a1: m ∈ set b
  assume x2 ∈ set m
  then have f2: atm-of x2 ∈ atms-of (mset m)
    by simp
  have  $\bigwedge f. (f m :: 'v \text{ clause}) \in f \text{ ' set } b$ 
    using a1 by blast
  then have  $\bigwedge f. (\text{atms-of } (f m) :: 'v \text{ set}) \subseteq \text{atms-of-ms } (f \text{ ' set } b)$ 
    by simp
  then have  $\bigwedge n f. (n :: 'v) \in \text{atms-of-ms } (f \text{ ' set } b) \vee n \notin \text{atms-of } (f m)$ 
    by (meson contra-subsetD)
  then have atm-of x2 ∈ atms-of-ms (mset ' set b)
    using f2 by blast
} note H = this
{
  fix m :: 'v literal list and x2
  have  $m \in \text{set } b \implies x2 \in \text{set } m \implies x2 \notin \text{lits-of-l } a \implies \neg x2 \notin \text{lits-of-l } a \implies$ 
     $\exists aa \in \text{set } b. \neg \text{atm-of ' set } aa \subseteq \text{atm-of ' lits-of-l } a$ 
    by (meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI)
} note H' = this

assume do-decide-step S ≠ S and
  S = (a, b, c, d, e) and
  conflicting S = None
then show decide S (do-decide-step S)
  using H H' by (auto split: option.splits simp: decide.simps Marked-Propagated-in-iff-in-lits-of-l
    dest!: find-first-unused-var-Some)
qed

lemma mmset-of-mlit'-eq-Marked[iff]: mmset-of-mlit' z = Marked x k  $\longleftrightarrow$  z = Marked x k
  by (cases z) auto

lemma do-decide-step-no:
  do-decide-step S = S  $\implies$  no-step decide S
  apply (cases S, cases conflicting S)
  apply (auto simp: atms-of-ms-mset-unfold Marked-Propagated-in-iff-in-lits-of-l
    dest!: atm-of-in-atm-of-set-in-uminus
    elim!: decideE
    split: option.splits)+
  using atm-of-eq-atm-of by blast

lemma rough-state-of-state-of-do-decide-step[simp]:
  cdclW-all-struct-inv S  $\implies$  rough-state-of (state-of (do-decide-step S)) = do-decide-step S
proof (subst state-of-inverse, goal-cases)
  case 1
  then show ?case
    by (cases do-decide-step S = S)
      (auto dest: do-decide-step decide other intro: cdclW-all-struct-inv-inv)
qed simp

lemma rough-state-of-state-of-do-skip-step[simp]:
  cdclW-all-struct-inv S  $\implies$  rough-state-of (state-of (do-skip-step S)) = do-skip-step S
  apply (subst state-of-inverse, cases do-skip-step S = S)
  apply simp

```

by (blast dest: other skip bj do-skip-step cdcl_W-all-struct-inv-inv)+

19.3.3 Code generation

Type definition There are two invariants: one while applying conflict and propagate and one for the other rules

declare rough-state-of-inverse[simp add]

definition Con **where**

Con xs = state-of (if cdcl_W-all-struct-inv xs then xs else ([], [], [], 0, None))

lemma [code abstype]:

Con (rough-state-of S) = S

using rough-state-of[of S] **unfolding** Con-def **by** simp

definition do-cp-step' **where**

do-cp-step' S = state-of (do-cp-step (rough-state-of S))

typedef 'v cdcl_W-state-inv-from-init-state = {S::'v cdcl_W-state-inv-st. cdcl_W-all-struct-inv S
 \wedge cdcl_W-stgy** (raw-S0-cdcl_W (raw-init-clss S)) S}

morphisms rough-state-from-init-state-of state-from-init-state-of

proof

show ([], [], [], 0, None) \in {S. cdcl_W-all-struct-inv S
 \wedge cdcl_W-stgy** (raw-S0-cdcl_W (raw-init-clss S)) S}

by (auto simp add: cdcl_W-all-struct-inv-def)

qed

instantiation cdcl_W-state-inv-from-init-state :: (type) equal

begin

definition equal-cdcl_W-state-inv-from-init-state :: 'v cdcl_W-state-inv-from-init-state \Rightarrow

'v cdcl_W-state-inv-from-init-state \Rightarrow bool **where**

equal-cdcl_W-state-inv-from-init-state S S' \longleftrightarrow

(rough-state-from-init-state-of S = rough-state-from-init-state-of S')

instance

by standard (simp add: rough-state-from-init-state-of-inject

equal-cdcl_W-state-inv-from-init-state-def)

end

definition ConI **where**

ConI S = state-from-init-state-of (if cdcl_W-all-struct-inv S

\wedge cdcl_W-stgy** (raw-S0-cdcl_W (raw-init-clss S)) S then S else ([], [], [], 0, None))

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:: 'v cdcl_W-state-inv-from-init-state \Rightarrow 'v cdcl_W-state-inv **where**

id-of-I-to S = state-of (rough-state-from-init-state-of S)

lemma [code abstract]:

rough-state-of (id-of-I-to S) = rough-state-from-init-state-of S

unfolding id-of-I-to-def **using** rough-state-from-init-state-of[of S] **by** auto

Conflict and Propagate function do-full1-cp-step :: 'v cdcl_W-state-inv \Rightarrow 'v cdcl_W-state-inv
where

```

do-full1-cp-step S =
  (let S' = do-cp-step' S in
    if S = S' then S else do-full1-cp-step S')
by auto
termination
proof (relation {(T', T). (rough-state-of T', rough-state-of T) ∈ {(S', S).
  (S', S) ∈ {(S', S). cdclW-all-struct-inv S ∧ cdclW-cp S S'}}, goal-cases)
  case 1
  show ?case
    using wf-if-measure-f[OF wf-if-measure-f[OF cdclW-cp-wf-all-inv, of ], of rough-state-of] .
next
  case (2 S' S)
  then show ?case
    unfolding do-cp-step'-def
    apply simp
    by (metis cp-step-is-cdclW-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 λS. 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 ∈ set (raw-init-clss (rough-state-of S) @ raw-learned-clss (rough-state-of S)) ⇒ distinct c
apply (cases rough-state-of S)
using rough-state-of[of S] by (auto simp add: distinct-mset-set-distinct cdclW-all-struct-inv-def
  distinct-cdclW-state-def)

lemma do-full1-cp-step-full:
  full cdclW-cp (rough-state-of S)
  (rough-state-of (do-full1-cp-step S))
  unfolding full-def
proof (rule conjI, induction S rule: do-full1-cp-step.induct)
  case (1 S)
  then have f1:
    cdclW-cp** ((do-cp-step (rough-state-of S))) (
      rough-state-of (do-full1-cp-step (state-of (do-cp-step (rough-state-of S)))))
    ∨ state-of (do-cp-step (rough-state-of S)) = S
  using rough-state-of-state-of-do-cp-step[of S] unfolding do-cp-step'-def by fastforce
  have f2: ∧c. (if c = state-of (do-cp-step (rough-state-of c))
    then c else do-full1-cp-step (state-of (do-cp-step (rough-state-of c))))
    = do-full1-cp-step c
  by (metis (full-types) do-cp-step'-def do-full1-cp-step.simps)
  have f3: ¬ cdclW-cp (rough-state-of S) (do-cp-step (rough-state-of S))
    ∨ state-of (do-cp-step (rough-state-of S)) = S
    ∨ cdclW-cp++ (rough-state-of S)
      (rough-state-of (do-full1-cp-step (state-of (do-cp-step (rough-state-of S)))))
  using f1 by (meson rtranclp-into-tranclp2)
  { assume do-full1-cp-step S ≠ S
  then have do-cp-step (rough-state-of S) = rough-state-of S
    → cdclW-cp** (rough-state-of S) (rough-state-of (do-full1-cp-step S))
    ∨ do-cp-step (rough-state-of S) ≠ rough-state-of S
    ∧ state-of (do-cp-step (rough-state-of S)) ≠ S
  }

```

```

    using f2 f1 by (metis (no-types))
  then have do-cp-step (rough-state-of S)  $\neq$  rough-state-of S
     $\wedge$  state-of (do-cp-step (rough-state-of S))  $\neq$  S
     $\vee$  cdclW-cp** (rough-state-of S) (rough-state-of (do-full1-cp-step S))
    by (metis rough-state-of-state-of-do-cp-step)
  then have cdclW-cp** (rough-state-of S) (rough-state-of (do-full1-cp-step S))
    using f3 f2 by (metis (no-types) cp-step-is-cdclW-cp tranclp-into-rtranclp) }
then show ?case
  by fastforce
next
show no-step cdclW-cp (rough-state-of (do-full1-cp-step S))
  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
qed

```

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 S and
 st: do-other-step S \neq S
 shows cdcl_W-o S (do-other-step S)
 using st inv by (auto split: if-split-asm
 simp add: Let-def
 intro: do-skip-step do-resolve-step do-backtrack-step do-decide-step
 cdcl_W-o.intros cdcl_W-bj.intros)

lemma do-other-step-no:
 assumes inv: cdcl_W-all-struct-inv S and
 st: do-other-step S = S
 shows no-step cdcl_W-o S
 using st inv by (auto split: if-split-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  $cdcl_W-o$  ( $rough-state-of\ S$ ) ( $do-other-step\ (rough-state-of\ S)$ )
  by ( $metis\ False\ cdcl_W-all-struct-inv-rough-state\ do-other-step[of\ rough-state-of\ S]$ )
then have  $cdcl_W-all-struct-inv$  ( $do-other-step\ (rough-state-of\ S)$ )
  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** ($auto\ intro:\ cdcl_W-all-struct-inv-inv$
 $cdcl_W-all-struct-inv-rough-state\ other\ state-of-inverse$)

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 $toS-do-full1-cp-step-not-eq$: $do-full1-cp-step\ S \neq S \implies$

```

 $rough-state-of\ S \neq rough-state-of\ (do-full1-cp-step\ S)$ 

```

proof –

assume $a1$: $do-full1-cp-step\ S \neq S$

then have $S \neq do-cp-step'\ S$

by $fastforce$

then show $?thesis$

by ($metis\ (no-types)\ do-cp-step'-def\ do-full1-cp-step-fix-point-of-do-full1-cp-step$
 $rough-state-of-inverse$)

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 $cdcl_W-stgy\ (rough-state-of\ S)\ (rough-state-of\ (do-cdcl_W-stgy-step\ S))$

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  $cdcl_W\text{-}o$  (rough-state-of  $S$ ) (rough-state-of (do-other-step'  $S$ ))
  by (smt True assms  $cdcl_W\text{-all-struct-inv-rough-state}$   $do\text{-}cdcl_W\text{-stgy-step-def}$  do-other-step
    rough-state-of-do-other-step' rough-state-of-inverse)
moreover
  have
    np: no-step propagate (rough-state-of  $S$ ) and
    nc: no-step conflict (rough-state-of  $S$ )
    apply (metis True  $cdcl_W\text{-cp.simps}$  do-cp-step-eq-no-step
      do-full1-cp-step-fix-point-of-do-full1-cp-step in-clauses-rough-state-of-is-distinct)
    by (metis True do-conflict-step-no-step do-cp-step-eq-no-prop-no-confl
      do-full1-cp-step-fix-point-of-do-full1-cp-step)
    then have no-step  $cdcl_W\text{-cp}$  (rough-state-of  $S$ )
    by (simp add:  $cdcl_W\text{-cp.simps}$ )
  moreover have full  $cdcl_W\text{-cp}$  (rough-state-of (do-other-step'  $S$ ))
    (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  $do\text{-}cdcl_W\text{-stgy-step-def}$ 
    by (auto intro!:  $cdcl_W\text{-stgy.other'}$  dest:  $toS\text{-do-full1-cp-step-not-eq}$ )
qed

```

lemma *do-skip-step-trail-changed-or-conflict*:

```

assumes d: do-other-step  $S \neq S$ 
and inv:  $cdcl_W\text{-all-struct-inv}$   $S$ 
shows trail  $S \neq \text{trail}$  (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  $cdcl_W\text{-}M\text{-level-inv}$   $S$ 
  using inv unfolding  $cdcl_W\text{-all-struct-inv-def}$  by auto

```

```

have  $cdcl_W\text{-}o$   $S$  (do-other-step  $S$ ) using  $do\text{-other-step}[OF\ inv\ d]$  .

```

```

then show ?thesis

```

```

  using  $\langle cdcl_W\text{-}M\text{-level-inv } S \rangle$ 

```

```

proof (induction do-other-step  $S$  rule:  $cdcl_W\text{-}o\text{-induct-lev2}$ )

```

```

  case decide

```

```

    then show ?thesis

```

```

      apply (cases  $S$ )

```

```

      apply (auto dest!: find-first-unused-var-Some
        simp: split: option.splits)

```

```

      by (meson atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set contra-subsetD)

```

```

  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  $U = \text{this}(7)$ 

```

```

    then show ?case

```

```

      apply (cases do-other-step  $S$ )

```

```

      apply (auto split: if-split-asm simp: Let-def)

```

```

    apply (cases S rule: do-skip-step.cases; auto split: if-split-asm)
  apply (cases S rule: do-skip-step.cases; auto split: if-split-asm)

  apply (cases S rule: do-backtrack-step.cases;
    auto split: if-split-asm option.splits list.splits marked-lit.splits
    dest!: bt-cut-some-decomp simp: Let-def)
  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:
   $\exists c. \text{raw-trail} (\text{do-cp-step } S) = c @ \text{raw-trail } S \wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$ 
  by (cases S, cases raw-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:
   $\exists c. \text{raw-trail} (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{raw-trail} (\text{rough-state-of } S)$ 
   $\wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$ 
  apply (induction rule: do-full1-cp-step-induct)
  apply (rename-tac S, case-tac  $\text{do-cp-step}' S = S$ )
  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:
   $\text{conflicting} (\text{rough-state-of } S) \neq \text{None} \implies \text{do-cp-step}' S = S$ 
  unfolding do-cp-step'-def do-cp-step-def by simp

lemma do-full1-cp-step-conflicting:
   $\text{conflicting} (\text{rough-state-of } S) \neq \text{None} \implies \text{do-full1-cp-step } S = S$ 
  unfolding do-cp-step'-def do-cp-step-def
  apply (induction rule: do-full1-cp-step-induct)
  by (rename-tac S, case-tac  $S \neq \text{do-cp-step}' S$ )
  (auto simp add: do-full1-cp-step.simps do-cp-step-conflicting)

lemma do-decide-step-not-conflicting-one-more-decide:
  assumes
     $\text{conflicting } S = \text{None}$  and
     $\text{do-decide-step } S \neq S$ 
  shows  $\text{Suc } (\text{length } (\text{filter is-marked } (\text{raw-trail } S)))$ 
     $= \text{length } (\text{filter is-marked } (\text{raw-trail } (\text{do-decide-step } S)))$ 
  using assms unfolding do-other-step'-def
  by (cases S) (force simp: Let-def split: if-split-asm option.splits
    dest!: find-first-unused-var-Some-not-all-incl)

lemma do-decide-step-not-conflicting-one-more-decide-bt:
  assumes  $\text{conflicting } S \neq \text{None}$  and
     $\text{do-decide-step } S \neq S$ 
  shows  $\text{length } (\text{filter is-marked } (\text{raw-trail } S)) <$ 
     $\text{length } (\text{filter is-marked } (\text{raw-trail } (\text{do-decide-step } S)))$ 

```

using *assms* **unfolding** *do-other-step'-def* **by** (*cases S*, *cases conflicting S*)
(auto simp add: Let-def split: if-split-asm option.splits)

lemma *do-other-step-not-conflicting-one-more-decide-bt*:

assumes

conflicting (rough-state-of S) ≠ None **and**

conflicting (rough-state-of (do-other-step' S)) = None **and**

do-other-step' S ≠ S

shows *length (filter is-marked (raw-trail (rough-state-of S)))*

> length (filter is-marked (raw-trail (rough-state-of (do-other-step' S))))

proof (*cases S*, *goal-cases*)

case (*1 y*) **note** *S = this(1)* **and** *inv = this(2)*

obtain *M N U k E* **where** *y: y = (M, N, U, k, Some E)*

using *assms(1)* *S inv* **by** (*cases y*, *cases conflicting y*) *auto*

have *M*: *rough-state-of (state-of (M, N, U, k, Some E)) = (M, N, U, k, Some E)*

using *inv y* **by** (*auto simp add: state-of-inverse*)

have *bt*: *do-other-step' S = state-of (do-backtrack-step (rough-state-of S))*

proof (*cases rough-state-of S rule: do-decide-step.cases*)

case *1*

then show *?thesis*

using *assms(1,2)* **by** *auto*[]

next

case (*2 v vb vd vf vh*)

have *f3*: $\bigwedge c. (if\ do-skip-step\ (rough-state-of\ c) \neq rough-state-of\ c$

then do-skip-step (rough-state-of c)

else if do-resolve-step (do-skip-step (rough-state-of c)) ≠ do-skip-step (rough-state-of c)

then do-resolve-step (do-skip-step (rough-state-of c))

else if do-backtrack-step (do-resolve-step (do-skip-step (rough-state-of c)))

≠ do-resolve-step (do-skip-step (rough-state-of c))

then do-backtrack-step (do-resolve-step (do-skip-step (rough-state-of c)))

else do-decide-step (do-backtrack-step (do-resolve-step

(do-skip-step (rough-state-of c))))

= rough-state-of (do-other-step' c)

by (*simp add: rough-state-of-do-other-step'*)

have

(raw-trail (rough-state-of (do-other-step' S)),

raw-init-clss (rough-state-of (do-other-step' S)),

raw-learned-clss (rough-state-of (do-other-step' S)),

raw-backtrack-lvl (rough-state-of (do-other-step' S)), None)

= rough-state-of (do-other-step' S)

using *assms(2)* **by** (*cases do-other-step' S*) *auto*

then show *?thesis*

using *f3 2* **by** (*metis (no-types) do-decide-step.simps(2) do-resolve-step-trail-is-None*

do-skip-step-trail-is-None rough-state-of-inverse)

qed

show *?case*

using *assms(2)* *S* **unfolding** *bt y inv*

apply *simp*

by (*auto simp add: M bt-cut-not-none*

split: option.splits

dest!: bt-cut-some-decomp)

qed

lemma *do-other-step-not-conflicting-one-more-decide*:

assumes *conflicting (rough-state-of S) = None* **and**

do-other-step' S \neq *S*
shows $1 + \text{length} (\text{filter } \text{is-marked} (\text{raw-trail} (\text{rough-state-of } S)))$
 $= \text{length} (\text{filter } \text{is-marked} (\text{raw-trail} (\text{rough-state-of } (\text{do-other-step' } S))))$
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** $y: y = (M, N, U, k, \text{None})$ **using** $\text{assms}(1)\ S\ \text{inv}$ **by** (*cases y*) *auto*
have $M: \text{rough-state-of } (\text{state-of } (M, N, U, k, \text{None})) = (M, N, U, k, \text{None})$
using $\text{inv}\ y$ **by** (*auto simp add: state-of-inverse*)
have $\text{state-of } (\text{do-decide-step } (M, N, U, k, \text{None})) \neq \text{state-of } (M, N, U, k, \text{None})$
using $\text{assms}(2)$ **unfolding** $\text{do-other-step'-def } y\ \text{inv } S$ **by** (*auto simp add: M*)
then have $f4: \text{do-skip-step } (\text{rough-state-of } S) = \text{rough-state-of } S$
unfolding $S\ M\ y$ **by** (*metis (full-types) do-skip-step.simps(4)*)
have $f5: \text{do-resolve-step } (\text{rough-state-of } S) = \text{rough-state-of } S$
unfolding $S\ M\ y$ **by** (*metis (no-types) do-resolve-step.simps(4)*)
have $f6: \text{do-backtrack-step } (\text{rough-state-of } S) = \text{rough-state-of } S$
unfolding $S\ M\ y$ **by** (*metis (no-types) do-backtrack-step.simps(2)*)
have $\text{do-other-step } (\text{rough-state-of } S) \neq \text{rough-state-of } S$
using $\text{assms}(2)$ **unfolding** $S\ M\ y\ \text{do-other-step'-def}$ **by** (*metis (no-types)*)
then show *?case*
using $f6\ f5\ f4$ **by** (*simp add: assms(1) do-decide-step-not-conflicting-one-more-decide*
 $\text{do-other-step'-def}$)
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]:*
 $\text{conflicting } (\text{do-resolve-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
by (*cases S rule: do-resolve-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *conflicting-do-skip-step-iff[iff]:*
 $\text{conflicting } (\text{do-skip-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
by (*cases S rule: do-skip-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *conflicting-do-decide-step-iff[iff]:*
 $\text{conflicting } (\text{do-decide-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
by (*cases S rule: do-decide-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *conflicting-do-backtrack-step-imp[simp]:*
 $\text{do-backtrack-step } S \neq S \implies \text{conflicting } (\text{do-backtrack-step } S) = \text{None}$
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*:
 $do-backtrack-step\ S = S \longleftrightarrow raw-trail\ (do-backtrack-step\ S) = raw-trail\ S$
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*:
 $do-resolve-step\ S = S \longleftrightarrow trail\ (do-resolve-step\ S) = trail\ S$
by (cases *S* rule: *do-resolve-step.cases*) auto

lemma *do-other-step-eq-iff-trail-eq*:
 $do-other-step\ S = S \longleftrightarrow raw-trail\ (do-other-step\ S) = raw-trail\ S$

apply
(auto simp add: Let-def *do-skip-step-eq-iff-trail-eq*
do-decide-step-eq-iff-trail-eq *do-backtrack-step-eq-iff-trail-eq*
do-resolve-step-eq-iff-trail-eq
)
apply (simp add: *do-resolve-step-eq-iff-trail-eq[symmetric]*
do-skip-step-eq-iff-trail-eq[symmetric])
apply (simp add: *do-skip-step-eq-iff-trail-eq[symmetric]*
do-decide-step-eq-iff-trail-eq *do-backtrack-step-eq-iff-trail-eq[symmetric]*
do-resolve-step-eq-iff-trail-eq[symmetric]
)
done

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 None
then have *tr*: $trail\ (rough-state-of\ (do-full1-cp-step\ ?T)) = trail\ (rough-state-of\ ?T)$
using *do-full1-cp-step-conflicting* **by** *fastforce*
have $raw-trail\ (rough-state-of\ (do-full1-cp-step\ (do-other-step'\ S))) =$
 $raw-trail\ (rough-state-of\ S)$
using *arg-cong[OF H, of $\lambda S. raw-trail\ (rough-state-of\ S)$]* .
then have $raw-trail\ (rough-state-of\ (do-other-step'\ S)) = raw-trail\ (rough-state-of\ S)$
using *confl* **by** (auto simp add: *do-full1-cp-step-conflicting*)
then have $do-other-step'\ S = S$
by (simp add: *do-other-step-eq-iff-trail-eq[symmetric]* *do-other-step'-def*
del: do-other-step.simps)
}

moreover {
assume *eq[simp]*: $do-other-step'\ S = S$
obtain *c* **where** *c*: $raw-trail\ (rough-state-of\ (do-full1-cp-step\ S)) =$
 $c\ @\ raw-trail\ (rough-state-of\ S)$
using *do-full1-cp-step-neq-trail-increase* **by** auto

moreover have $raw-trail\ (rough-state-of\ (do-full1-cp-step\ S)) = raw-trail\ (rough-state-of\ S)$
using *arg-cong[OF H, of $\lambda S. raw-trail\ (rough-state-of\ S)$]* **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) = None and neg: do-other-step' S ≠ S
obtain c where
  c: raw-trail (rough-state-of (do-full1-cp-step ?T)) = c @ raw-trail (rough-state-of ?T) and
  nm:  $\forall m \in \text{set } c. \neg \text{is-marked } m$ 
  using do-full1-cp-step-neg-trail-increase by auto
have length (filter is-marked (raw-trail (rough-state-of (do-full1-cp-step ?T))))
  = length (filter is-marked (raw-trail (rough-state-of ?T)))
  using nm unfolding c by force
moreover have length (filter is-marked (raw-trail (rough-state-of S)))
  ≠ length (filter is-marked (raw-trail (rough-state-of ?T)))
  using do-other-step-not-conflicting-one-more-decide[OF - neg]
  do-other-step-not-conflicting-one-more-decide-bt[of S, OF - confl neg]
  by linarith
finally have False unfolding H by blast
}
ultimately show ?thesis by blast
qed

```

```

lemma do-cdclW-stgy-step-no:
  assumes S: do-cdclW-stgy-step S = S
  shows no-step cdclW-stgy (rough-state-of S)
proof –
{
  fix S'
  assume full1 cdclW-cp (rough-state-of S) S'
  then have False
    using do-full1-cp-step-full[of S] unfolding full-def S rtranclp-unfold full1-def
    by (smt assms do-cdclW-stgy-step-def tranclpD)
}
moreover {
  fix S' S''
  assume cdclW-o (rough-state-of S) S' and
  no-step propagate (rough-state-of S) and
  no-step conflict (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]:
  rough-state-of (state-of (rough-state-from-init-state-of S))
  = 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-rtranclp-cdclW: cdclW-cp S T  $\implies$  cdclW** S T
apply (induction rule: cdclW-cp.induct)
using conflict apply blast
using propagate by blast

```

```

lemma rtranclp-cdclW-cp-is-rtranclp-cdclW: cdclW-cp** S T  $\implies$  cdclW** S T
apply (induction rule: rtranclp-induct)

```

apply *simp*
by (*fastforce dest!*: *cdcl_W-cp-is-rtrancpl-cdcl_W*)

lemma *cdcl_W-stgy-is-rtrancpl-cdcl_W*:
*cdcl_W-stgy S T \implies cdcl_W** S T*
apply (*induction rule*: *cdcl_W-stgy.induct*)
using *cdcl_W-stgy.conflict'* *rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W* **apply** *blast*
unfolding *full-def* **by** (*fastforce dest!*:*other rtrancpl-cdcl_W-cp-is-rtrancpl-cdcl_W*)

lemma *cdcl_W-stgy-init-clss*: *cdcl_W-stgy S T \implies cdcl_W-M-level-inv S \implies init-clss S = init-clss T*
using *rtrancpl-cdcl_W-init-clss cdcl_W-stgy-is-rtrancpl-cdcl_W* **by** *fast*

lemma *clauses-toS-rough-state-of-do-cdcl_W-stgy-step[*simp*]*:
init-clss (rough-state-of (do-cdcl_W-stgy-step (state-of (rough-state-from-init-state-of S))))
= init-clss (rough-state-from-init-state-of S) (is - = init-clss ?S)

proof (*cases do-cdcl_W-stgy-step (state-of ?S) = state-of ?S*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have $\bigwedge c. \text{cdcl}_W\text{-M-level-inv (rough-state-of } c)$
using *cdcl_W-all-struct-inv-def cdcl_W-all-struct-inv-rough-state* **by** *blast*
then have $\bigwedge c. \text{init-clss (rough-state-of } c) = \text{init-clss (rough-state-of (do-cdcl}_W\text{-stgy-step } c))$
 $\vee \text{do-cdcl}_W\text{-stgy-step } c = c$
using *cdcl_W-stgy-no-more-init-clss do-cdcl_W-stgy-step* **by** *blast*
then show *?thesis*
using *False* **by** *force*
qed

lemma *raw-init-clss-do-cp-step[*simp*]*:
raw-init-clss (do-cp-step S) = raw-init-clss S
by (*cases S*) (*auto simp: do-cp-step-def do-propagate-step-def do-conflict-step-def*
split: option.splits)

lemma *raw-init-clss-do-cp-step'[*simp*]*:
raw-init-clss (rough-state-of (do-cp-step' S)) = raw-init-clss (rough-state-of S)
by (*simp add: do-cp-step'-def*)

lemma *raw-init-clss-rough-state-of-do-full1-cp-step[*simp*]*:
raw-init-clss (rough-state-of (do-full1-cp-step S))
= raw-init-clss (rough-state-of S)
apply (*rule do-full1-cp-step.induct[of $\lambda S.$*
raw-init-clss (rough-state-of (do-full1-cp-step S))
= raw-init-clss (rough-state-of S)])
by (*metis (mono-tags, lifting) do-full1-cp-step.simps raw-init-clss-do-cp-step'*)

lemma *raw-init-clss-do-skip-def[*simp*]*:
raw-init-clss (do-skip-step S) = raw-init-clss S
by (*cases S rule: do-skip-step.cases*) (*auto simp: do-other-step'-def Let-def*
split: option.splits)

lemma *raw-init-clss-do-resolve-def[*simp*]*:
raw-init-clss (do-resolve-step S) = raw-init-clss S
by (*cases S rule: do-resolve-step.cases*) (*auto simp: do-other-step'-def Let-def*
split: option.splits)

lemma *raw-init-clss-do-backtrack-def*[simp]:
raw-init-clss (do-backtrack-step S) = raw-init-clss S
by (cases *S* rule: *do-backtrack-step.cases*) (auto simp: *do-other-step'-def Let-def*
split: option.splits list.splits marked-lit.splits)

lemma *raw-init-clss-do-decide-def*[simp]:
raw-init-clss (do-decide-step S) = raw-init-clss S
by (cases *S* rule: *do-decide-step.cases*) (auto simp: *do-other-step'-def Let-def*
split: option.splits)

lemma *raw-init-clss-rough-state-of-do-other-step'*[simp]:
raw-init-clss (rough-state-of (do-other-step' S))
= raw-init-clss (rough-state-of S)
by (cases *S*) (auto simp: *do-other-step'-def Let-def do-skip-step.cases*
split: option.splits)

lemma [simp]:
raw-init-clss (rough-state-of (do-cdcl_W-stgy-step (state-of (rough-state-from-init-state-of S))))
=
raw-init-clss (rough-state-from-init-state-of S)
unfolding *do-cdcl_W-stgy-step-def* **by** (auto simp: *Let-def*)

lemma *rough-state-from-init-state-of-do-cdcl_W-stgy-step'*[code abstract]:
rough-state-from-init-state-of (do-cdcl_W-stgy-step' S) =
rough-state-of (do-cdcl_W-stgy-step (id-of-I-to S))

proof –
let *?S* = (*rough-state-from-init-state-of S*)
have *cdcl_W-stgy*** (*raw-S0-cdcl_W (raw-init-clss (rough-state-from-init-state-of S))*)
(*rough-state-from-init-state-of S*)
using *rough-state-from-init-state-of*[of *S*] **by** auto
moreover have *cdcl_W-stgy***
(*rough-state-from-init-state-of S*)
(*rough-state-of (do-cdcl_W-stgy-step*
(*state-of (rough-state-from-init-state-of S))*)))
using *do-cdcl_W-stgy-step*[of *state-of ?S*]
by (cases *do-cdcl_W-stgy-step (state-of ?S) = state-of ?S*) auto
ultimately show *?thesis*
unfolding *do-cdcl_W-stgy-step'-def id-of-I-to-def*
by (auto intro: *state-from-init-state-of-inverse*)
qed

All rules together **function** *do-all-cdcl_W-stgy* **where**

do-all-cdcl_W-stgy S =
(*let T = do-cdcl_W-stgy-step' S in*
if T = S then S else do-all-cdcl_W-stgy T)

by *fast+*

termination

proof (*relation {(T, S).*
(*cdcl_W-measure (rough-state-from-init-state-of T),*
cdcl_W-measure (rough-state-from-init-state-of S))
∈ le_{rn} {(a, b). a < b} 3}, *goal-cases*)
case 1
show *?case* **by** (*rule wf-if-measure-f*) (auto intro!: *wf-le_{rn} wf-less*)
next

```

case (2 S T) note T = this(1) and ST = this(2)
let ?S = rough-state-from-init-state-of S
have S: cdclW-stgy** (raw-S0-cdclW (raw-init-clss ?S)) ?S
  using rough-state-from-init-state-of[of S] by auto
moreover have cdclW-stgy (rough-state-from-init-state-of S)
  (rough-state-from-init-state-of T)

proof –
  have  $\bigwedge c.$  rough-state-of (state-of (rough-state-from-init-state-of c)) =
    rough-state-from-init-state-of c
    using rough-state-from-init-state-of by force
  then have do-cdclW-stgy-step (state-of (rough-state-from-init-state-of S))
     $\neq$  state-of (rough-state-from-init-state-of S)
    using ST T rough-state-from-init-state-of-inverse
    unfolding id-of-I-to-def do-cdclW-stgy-step'-def
    by fastforce
  from do-cdclW-stgy-step[OF this] show ?thesis
    by (simp add: T id-of-I-to-def rough-state-from-init-state-of-do-cdclW-stgy-step')
qed
moreover
  have cdclW-all-struct-inv (rough-state-from-init-state-of S)
    using rough-state-from-init-state-of[of S] by auto
  then have cdclW-all-struct-inv (raw-S0-cdclW (raw-init-clss (rough-state-from-init-state-of S)))
    by (cases rough-state-from-init-state-of S)
    (auto simp add: cdclW-all-struct-inv-def distinct-cdclW-state-def)
  ultimately show ?case
    by (auto intro!: cdclW-stgy-step-decreasing[of - - raw-S0-cdclW (raw-init-clss ?S)]
      simp del: cdclW-measure.simps)
qed

thm do-all-cdclW-stgy.induct
lemma do-all-cdclW-stgy-induct:
  ( $\bigwedge S. (do-cdcl_W-stgy-step' S \neq S \implies P (do-cdcl_W-stgy-step' S)) \implies P S \implies P a0$ )
  using do-all-cdclW-stgy.induct by metis

lemma [simp]: raw-init-clss (rough-state-from-init-state-of (do-all-cdclW-stgy S)) =
  raw-init-clss (rough-state-from-init-state-of S)
apply (induction rule: do-all-cdclW-stgy-induct)
by (smt do-all-cdclW-stgy.simps do-cdclW-stgy-step-def id-of-I-to-def
  raw-init-clss-rough-state-of-do-full1-cp-step raw-init-clss-rough-state-of-do-other-step'
  rough-state-from-init-state-of-do-cdclW-stgy-step'
  toS-rough-state-of-state-of-rough-state-from-init-state-of)

lemma no-step-cdclW-stgy-cdclW-all:
  fixes S :: 'a cdclW-state-inv-from-init-state
  shows no-step cdclW-stgy (rough-state-from-init-state-of (do-all-cdclW-stgy S))
  apply (induction S rule: do-all-cdclW-stgy-induct)
  apply (rename-tac S, case-tac do-cdclW-stgy-step' S  $\neq$  S)
proof –
  fix Sa :: 'a cdclW-state-inv-from-init-state
  assume a1:  $\neg do-cdcl_W-stgy-step' Sa \neq Sa$ 
  { fix pp
    have (if True then Sa else do-all-cdclW-stgy Sa) = do-all-cdclW-stgy Sa
      using a1 by auto
    then have  $\neg cdcl_W-stgy (rough-state-from-init-state-of (do-all-cdcl_W-stgy Sa))$  pp
  }

```

```

    using a1 by (smt 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 (rough-state-from-init-state-of (do-all-cdclW-stgy Sa))
    by fastforce
next
fix Sa :: 'a cdclW-state-inv-from-init-state
assume a1: do-cdclW-stgy-step' Sa ≠ Sa
  ⇒ no-step cdclW-stgy (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 (rough-state-from-init-state-of (do-all-cdclW-stgy Sa))
  using a2 a1 by presburger
qed

lemma do-all-cdclW-stgy-is-rtranclp-cdclW-stgy:
  cdclW-stgy** (rough-state-from-init-state-of S)
    (rough-state-from-init-state-of (do-all-cdclW-stgy S))
proof (induction S rule: do-all-cdclW-stgy-induct)
case (1 S) note IH = this(1)
show ?case
proof (cases do-cdclW-stgy-step' S = S)
case True
  then show ?thesis by simp
next
case False
  have f2: do-cdclW-stgy-step (id-of-I-to S) = id-of-I-to S ⟶
    rough-state-from-init-state-of (do-cdclW-stgy-step' S)
    = rough-state-of (state-of (rough-state-from-init-state-of S))
    unfolding rough-state-from-init-state-of-do-cdclW-stgy-step'
      id-of-I-to-def by presburger
  have f3: do-all-cdclW-stgy S = do-all-cdclW-stgy (do-cdclW-stgy-step' S)
    by (metis (full-types) do-all-cdclW-stgy.simps)
  have cdclW-stgy (rough-state-from-init-state-of S)
    (rough-state-from-init-state-of (do-cdclW-stgy-step' S))
    = cdclW-stgy (rough-state-of (id-of-I-to S))
    (rough-state-of (do-cdclW-stgy-step (id-of-I-to S)))
    unfolding 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 by presburger
  then show ?thesis
    using f3 f2 IH do-cdclW-stgy-step
    by (smt False toS-rough-state-of-state-of-rough-state-from-init-state-of tranclp.intros(1)
      tranclp-into-rtranclp transitive-closurep-trans'(2))
qed
qed

```

Final theorem:

```

lemma consistent-interp-mmset-of-mlit[simp]:
  consistent-interp (lit-of ' mmset-of-mlit' ' set M') ⟷
  consistent-interp (lit-of ' set M')
by (auto simp: image-image)

```

```

lemma DPLL-tot-correct:
  assumes

```

```

  r: rough-state-from-init-state-of (do-all-cdclW-stgy (state-from-init-state-of
    ([], map remdups N, [], 0, None)))) = S and
  S: (M', N', U', k, E) = S
shows (E ≠ Some [] ∧ satisfiable (set (map mset N)))
  ∨ (E = Some [] ∧ unsatisfiable (set (map mset N)))
proof -
  let ?N = map remdups N
  have inv: cdclW-all-struct-inv ([], map remdups N, [], 0, None)
    unfolding cdclW-all-struct-inv-def distinct-cdclW-state-def distinct-mset-set-def by auto
  then have S0: rough-state-of (state-of ([], map remdups N, [], 0, None))
    = ([], map remdups N, [], 0, None) by simp
  have 1: full cdclW-stgy ([], ?N, [], 0, None) S
    unfolding full-def apply rule
    using do-all-cdclW-stgy-is-rtrancpl-cdclW-stgy[of
      state-from-init-state-of ([], map remdups N, [], 0, None)] inv
    by (auto simp del: do-all-cdclW-stgy.simps simp: state-from-init-state-of-inverse
      r[symmetric] no-step-cdclW-stgy-cdclW-all)+
  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 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-l M')
      unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def S[symmetric]
      by (auto simp: )
  moreover
    have [simp]:
      rough-state-from-init-state-of (state-from-init-state-of (raw-S0-cdclW (map remdups N)))
      = raw-S0-cdclW (map remdups N)
    apply (rule cdclW-state-inv-from-init-state.state-from-init-state-of-inverse)
    using 3 by (auto simp: cdclW-all-struct-inv-def distinct-cdclW-state-def
      image-image comp-def)
    have raw-init-clss ([], ?N, [], 0, None) = raw-init-clss S
      using arg-cong[OF r, of raw-init-clss] unfolding S[symmetric]
      by (simp del: do-all-cdclW-stgy.simps)
    then have N': N' = map remdups N
      using S[symmetric] by auto
  have conflicting S = Some {#} ∧ unsatisfiable (set-mset (init-clss S)) ∨
    conflicting S = None ∧ (case S of (M, uu-) ⇒ map mmset-of-mlit' M) ⊨asm init-clss S
  apply (rule full-cdclW-stgy-final-state-conclusive)
    using 1 apply simp
    using 2 apply simp
    using 3 by simp
  then have (E ≠ Some [] ∧ satisfiable (set (map mset ?N)))
    ∨ (E = Some [] ∧ unsatisfiable (set (map mset ?N)))
    using cons unfolding S[symmetric] N' apply (auto simp: comp-def)
    by (simp add: true-annots-true-clss)
  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. The only difference between the generated code and

the one used here is the export of the constructor `ConI`.

```

end
theory CDCL-W-Merge
imports CDCL-W-Termination
begin

```

20 Link between Weidenbach's and NOT's CDCL

20.1 Inclusion of the states

```

context conflict-driven-clause-learningW
begin
declare cdclW.intros[intro] cdclW-bj.intros[intro] cdclW-o.intros[intro]

lemma backtrack-no-cdclW-bj:
  assumes cdcl: cdclW-bj T U and inv: cdclW-M-level-inv V
  shows  $\neg$ backtrack V T
  using cdcl inv
  apply (induction rule: cdclW-bj.induct)
    apply (elim skipE, force elim!: backtrack-levE[OF - inv] simp: cdclW-M-level-inv-def)
    apply (elim resolveE, force elim!: backtrack-levE[OF - inv] simp: cdclW-M-level-inv-def)
  apply standard
  apply (elim backtrack-levE[OF - inv], elim backtrackE)
  apply (force simp del: state-simp simp add: state-eq-def cdclW-M-level-inv-decomp)
done

inductive skip-or-resolve :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool' where
s-or-r-skip[intro]: skip S T  $\Longrightarrow$  skip-or-resolve S T |
s-or-r-resolve[intro]: resolve S T  $\Longrightarrow$  skip-or-resolve S T

lemma rtranclp-cdclW-bj-skip-or-resolve-backtrack:
  assumes cdclW-bj** S U and inv: cdclW-M-level-inv S
  shows skip-or-resolve** S U  $\vee$  ( $\exists T$ . skip-or-resolve** S T  $\wedge$  backtrack T U)
  using assms
proof (induction)
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and bj = this(2) and IH = this(3)[OF this(4)]
  consider
    (SU) S = U
  | (SUp) cdclW-bj++ S U
  using st unfolding rtranclp-unfold by blast
  then show ?case
  proof cases
    case SUp
    have  $\bigwedge T$ . skip-or-resolve** S T  $\Longrightarrow$  cdclW** S T
    using mono-rtranclp[of skip-or-resolve cdclW]
    by (blast intro: other cdclW-o.intros cdclW-bj.intros skip-or-resolve.cases)
    then have skip-or-resolve** S U
    using bj IH inv backtrack-no-cdclW-bj rtranclp-cdclW-consistent-inv[OF - inv] by meson
    then show ?thesis
    using bj by (auto simp: cdclW-bj.simps dest!: skip-or-resolve.intros)
  next

```

```

    case SU
    then show ?thesis
      using bj by (auto simp: cdclW-bj.simps dest!: skip-or-resolve.intros)
    qed
  qed

```

```

lemma rtrancpl-skip-or-resolve-rtrancpl-cdclW:
  skip-or-resolve** S T  $\implies$  cdclW** S T
  by (induction rule: rtrancpl-induct)
  (auto dest!: cdclW-bj.intros cdclW.intros cdclW-o.intros simp: skip-or-resolve.simps)

```

definition *backjump-l-cond* :: '*v* clause \Rightarrow '*v* clause \Rightarrow '*v* literal \Rightarrow '*st* \Rightarrow '*st* \Rightarrow bool **where**
backjump-l-cond \equiv $\lambda C\ C'\ L'\ S\ T.$ True

definition *inv_{NOT}* :: '*st* \Rightarrow bool **where**
inv_{NOT} \equiv $\lambda S.$ no-dup (trail *S*)

```

declare invNOT-def[simp]
end

```

```

context conflict-driven-clause-learningW
begin

```

20.2 More lemmas conflict-propagate and backjumping

20.2.1 Termination

```

lemma cdclW-cp-normalized-element-all-inv:
  assumes inv: cdclW-all-struct-inv S
  obtains T where full cdclW-cp S T
  using assms cdclW-cp-normalized-element unfolding cdclW-all-struct-inv-def by blast
thm backtrackE

```

```

lemma cdclW-bj-measure:
  assumes cdclW-bj S T and cdclW-M-level-inv S
  shows length (trail S) + (if conflicting S = None then 0 else 1)
    > length (trail T) + (if conflicting T = None then 0 else 1)
  using assms by (induction rule: cdclW-bj.induct)
  (force dest:arg-cong[of - - length]
   intro: get-all-marked-decomposition-exists-prepend
   elim!: backtrack-levE skipE resolveE
   simp: cdclW-M-level-inv-def+)

```

```

lemma wf-cdclW-bj:
  wf {(b,a). cdclW-bj a b  $\wedge$  cdclW-M-level-inv a}
  apply (rule wfP-if-measure[of  $\lambda -.$  True
    -  $\lambda T.$  length (trail T) + (if conflicting T = None then 0 else 1), simplified])
  using cdclW-bj-measure by simp

```

```

lemma cdclW-bj-exists-normal-form:
  assumes lev: cdclW-M-level-inv S
  shows  $\exists T.$  full cdclW-bj S T
proof -
  obtain T where T: full ( $\lambda a\ b.$  cdclW-bj a b  $\wedge$  cdclW-M-level-inv a) S T
  using wf-exists-normal-form-full[OF wf-cdclW-bj] by auto
  then have cdclW-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  $cdcl_W$ ] by (blast intro:  $cdcl_W.intros$   $cdcl_W-o.intros$ )
  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
     $\exists M. trail\ S = M @ trail\ T \wedge (\forall m \in set\ M. \neg is-marked\ m)$ 
     $init-clss\ S = init-clss\ T$ 
     $learned-clss\ S = learned-clss\ T$ 
     $backtrack-lvl\ S = backtrack-lvl\ T$ 
  using assms by (induction rule: rtrancpl-induct)
  (auto simp del: state-simp simp: state-eq-def elim: skipE)

```

20.2.2 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  $cdcl_W$ -all-struct-inv  $V$ 
    using rtrancpl-mono[of skip  $cdcl_W$ ] assms(3) rtrancpl- $cdcl_W$ -all-struct-inv-inv mono-rtrancpl
    by (auto dest!: bj other  $cdcl_W$ -bj.skip)
  then have  $cdcl_W$ -M-level-inv  $V$ 
    unfolding  $cdcl_W$ -all-struct-inv-def by auto
  then obtain  $K\ i\ M1\ M2\ L\ D$  where
    conf: raw-conflicting  $V = Some\ D$  and
    LD:  $L \in \# mset-ccls\ D$  and
    decomp: (Marked  $K\ (Suc\ i) \# M1, M2) \in set\ (get-all-marked-decomposition\ (trail\ V))$  and
    lev-L: get-level (trail  $V$ )  $L = backtrack-lvl\ V$  and
    max: get-level (trail  $V$ )  $L = get-maximum-level\ (trail\ V)\ (mset-ccls\ D)$  and
    max-D: get-maximum-level (trail  $V$ ) (remove1-mset  $L\ (mset-ccls\ D)) \equiv i$  and
    undef: undefined-lit  $M1\ L$  and
    W:  $W \sim cons-trail\ (Propagated\ L\ (cls-of-ccls\ D))$ 
    (reduce-trail-to  $M1$ 
      (add-learned-cls (cls-of-ccls  $D$ )
        (update-backtrack-lvl  $i$ 
          (update-conflicting None  $V$ ))))
  using bt inv by (elim backtrack-levE) metis+
  obtain  $L'\ C'\ M\ E$  where
    tr: trail  $T = Propagated\ L'\ C'\ \# M$  and

```

```

raw: raw-conflicting  $T = \text{Some } E$  and
LE:  $-L' \notin \# \text{ mset-ccls } E$  and
E:  $\text{mset-ccls } E \neq \{\#\}$  and
V:  $V \sim \text{tl-trail } T$ 
using skip by (elim skipE) metis
let ?M = Propagated  $L' C' \# \text{ trail } V$ 
have tr-M: trail  $T = ?M$ 
using tr V by auto
have MT:  $M = \text{tl } (\text{trail } T)$  and MV:  $M = \text{trail } V$ 
using tr V by auto
have DE[simp]:  $\text{mset-ccls } D = \text{mset-ccls } E$ 
using V conf raw by (auto simp add: state-eq-def simp del: state-simp)
have cdclW** S T using bj cdclW-bj.skip mono-rtranclp[of skip cdclW S T] other st by meson
then have inv': cdclW-all-struct-inv T
using rtranclp-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 tr-M unfolding cdclW-M-level-inv-def by auto
let ?k = backtrack-lvl T
have [simp]:
backtrack-lvl V = ?k
using V by simp
have ?k > 0
using decomp M-lev V tr unfolding cdclW-M-level-inv-def by auto
then have atm-of L ∈ atm-of ' lits-of-l (trail V)
using lev-L get-rev-level-ge-0-atm-of-in[of 0 rev (trail V) L] by auto
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-l (trail V)
using n-d' unfolding lits-of-def by auto
have ?M ⊨as CNot (mset-ccls D)
using inv' raw unfolding cdclW-conflicting-def cdclW-all-struct-inv-def tr-M by auto
then have L' ∉ # mset-ccls (remove-clit L D)
using L-L' L'-M ⟨Propagated L' C' # trail V ⊨as CNot (mset-ccls D)⟩
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set unfolding true-annots-def
by (metis ⟨Propagated L' C' # trail V ⊨as CNot (mset-ccls D)⟩ in-remove1-mset-neq insert-iff
lits-of-l-cons marked-lit.sel(2) remove-lit true-annots-true-cls-def-iff-negation-in-model
uminus-not-id')
have [simp]: trail (reduce-trail-to M1 T) = M1
using decomp undef tr W V by auto
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 = init-clss V and [simp]: learned-clss S = learned-clss V
using rtranclp-skip-state-decomp[OF ⟨skip** S V⟩] V
by (auto simp del: state-simp simp: state-eq-def)
then have
W-S:  $W \sim \text{cons-trail } (\text{Propagated } L \text{ (cls-of-ccls } E)) \text{ (reduce-trail-to } M1$ 
 $(\text{add-learned-clss (cls-of-ccls } E) \text{ (update-backtrack-lvl } i \text{ (update-conflicting None } T))))$ 
using W V undef M-lev decomp tr
by (auto simp del: state-simp simp: state-eq-def cdclW-M-level-inv-def)

obtain M2' where
decomp':  $(\text{Marked } K \text{ (} i+1 \text{)} \# M1, M2') \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } T))$ 

```

```

using decomp V unfolding tr-M by (cases hd (get-all-marked-decomposition (trail V)),
  cases get-all-marked-decomposition (trail V)) auto
moreover
  from L-L' have get-level ?M L = ?k
    using lev-L V by (auto split: if-split-asm)
moreover
  have atm-of L'  $\notin$  atms-of (mset-ccls D)
    by (metis DE LE L-L'  $\langle$  L'  $\notin$  # mset-ccls (remove-clit L D)  $\rangle$ 
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def in-remove1-mset-neq remove-lit)
  then have get-level ?M L = get-maximum-level ?M (mset-ccls D)
    using calculation(2) lev-L max by auto
moreover
  have atm-of L'  $\notin$  atms-of (mset-ccls (remove-clit L D))
    by (metis DE LE  $\langle$  L'  $\notin$  # mset-ccls (remove-clit L D)  $\rangle$ 
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def in-remove1-mset-neq remove-lit
      in-atms-of-remove1-mset-in-atms-of)
  have i = get-maximum-level ?M (mset-ccls (remove-clit L D))
    using max-D  $\langle$  atm-of L'  $\notin$  atms-of (mset-ccls (remove-clit L D))  $\rangle$  by auto

ultimately have backtrack T W
  apply –
  apply (rule backtrack-rule[of T - L K i M1 M2' W, OF raw])
  unfolding tr-M[symmetric]
    using LD apply simp
    apply simp
    apply simp
    apply simp
    apply auto[]
  using W-S by auto
  then show ?thesis using IH inv by blast
qed

```

```

lemma fst-get-all-marked-decomposition-prepend-not-marked:
  assumes  $\forall m \in \text{set } MS. \neg \text{is-marked } m$ 
  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 (rename-tac L m xs; case-tac get-all-marked-decomposition (xs @ M)) simp-all

```

20.3 CDCL FW

```

inductive cdclW-merge-restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  fw-r-propagate: propagate S S'  $\Longrightarrow$  cdclW-merge-restart S S' |
  fw-r-conflict: conflict S T  $\Longrightarrow$  full cdclW-bj T U  $\Longrightarrow$  cdclW-merge-restart S U |
  fw-r-decide: decide S S'  $\Longrightarrow$  cdclW-merge-restart S S' |
  fw-r-rf: cdclW-rf S S'  $\Longrightarrow$  cdclW-merge-restart S S'

```

```

lemma rtrancp-cdclW-bj-rtrancp-cdclW:
  cdclW-bj** S T  $\Longrightarrow$  cdclW** S T
using mono-rtrancp[of cdclW-bj cdclW] by (blast intro: cdclW.other cdclW-o.bj)

```

```

lemma cdclW-merge-restart-cdclW:
  assumes cdclW-merge-restart S T
  shows cdclW** S T
  using assms

```

proof *induction*

case (*fw-r-conflict* $S\ T\ U$) **note** $\text{confl} = \text{this}(1)$ **and** $\text{bj} = \text{this}(2)$
have $\text{cdcl}_W\ S\ T$ **using** confl **by** (*simp* *add*: $\text{cdcl}_W.\text{intros}\ r\text{-into-rtrancp}$)
moreover
have $\text{cdcl}_W\text{-bj}^{**}\ T\ U$ **using** bj **unfolding** *full-def* **by** *auto*
then have $\text{cdcl}_W^{**}\ T\ U$ **using** $\text{rtrancp-cdcl}_W\text{-bj-rtrancp-cdcl}_W$ **by** *blast*
ultimately show *?case* **by** *auto*
qed (*simp-all* *add*: $\text{cdcl}_W\text{-o.intros}\ \text{cdcl}_W.\text{intros}\ r\text{-into-rtrancp}$)

lemma *cdcl_W-merge-restart-conflicting-true-or-no-step*:

assumes *cdcl_W-merge-restart* $S\ T$
shows *conflicting* $T = \text{None} \vee \text{no-step}\ \text{cdcl}_W\ T$
using *assms*

proof *induction*

case (*fw-r-conflict* $S\ T\ U$) **note** $\text{confl} = \text{this}(1)$ **and** $n\text{-s} = \text{this}(2)$
{ fix $D\ V$
assume $\text{cdcl}_W\ U\ V$ **and** *conflicting* $U = \text{Some}\ D$
then have *False*
using $n\text{-s}$ **unfolding** *full-def*
by (*induction rule*: $\text{cdcl}_W\text{-all-rules-induct}$)
(auto dest!: cdcl_W-bj.intros elim: decideE propagateE conflictE forgetE restartE)
}
then show *?case* **by** (*cases* *conflicting* U) *fastforce+*
qed (*auto simp add: cdcl_W-rf.simps elim: propagateE decideE restartE forgetE*)

inductive *cdcl_W-merge* :: $'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**

fw-propagate: $\text{propagate}\ S\ S' \Longrightarrow \text{cdcl}_W\text{-merge}\ S\ S' \mid$
fw-conflict: $\text{conflict}\ S\ T \Longrightarrow \text{full}\ \text{cdcl}_W\text{-bj}\ T\ U \Longrightarrow \text{cdcl}_W\text{-merge}\ S\ U \mid$
fw-decide: $\text{decide}\ S\ S' \Longrightarrow \text{cdcl}_W\text{-merge}\ S\ S' \mid$
fw-forget: $\text{forget}\ S\ S' \Longrightarrow \text{cdcl}_W\text{-merge}\ S\ S'$

lemma *cdcl_W-merge-cdcl_W-merge-restart*:

$\text{cdcl}_W\text{-merge}\ S\ T \Longrightarrow \text{cdcl}_W\text{-merge-restart}\ S\ T$
by (*meson* $\text{cdcl}_W\text{-merge.cases}\ \text{cdcl}_W\text{-merge-restart.simps}\ \text{forget}$)

lemma *rtrancp-cdcl_W-merge-trancp-cdcl_W-merge-restart*:

$\text{cdcl}_W\text{-merge}^{**}\ S\ T \Longrightarrow \text{cdcl}_W\text{-merge-restart}^{**}\ S\ T$
using $\text{rtrancp-mono}[\text{of}\ \text{cdcl}_W\text{-merge}\ \text{cdcl}_W\text{-merge-restart}]\ \text{cdcl}_W\text{-merge-cdcl}_W\text{-merge-restart}$ **by** *blast*

lemma *cdcl_W-merge-rtrancp-cdcl_W*:

$\text{cdcl}_W\text{-merge}\ S\ T \Longrightarrow \text{cdcl}_W^{**}\ S\ T$
using $\text{cdcl}_W\text{-merge-cdcl}_W\text{-merge-restart}\ \text{cdcl}_W\text{-merge-restart-cdcl}_W$ **by** *blast*

lemma *rtrancp-cdcl_W-merge-rtrancp-cdcl_W*:

$\text{cdcl}_W\text{-merge}^{**}\ S\ T \Longrightarrow \text{cdcl}_W^{**}\ S\ T$
using $\text{rtrancp-mono}[\text{of}\ \text{cdcl}_W\text{-merge}\ \text{cdcl}_W^{**}]\ \text{cdcl}_W\text{-merge-rtrancp-cdcl}_W$ **by** *auto*

lemmas *rulesE* =

skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE

lemma *cdcl_W-all-struct-inv-trancp-cdcl_W-merge-trancp-cdcl_W-merge-cdcl_W-all-struct-inv*:

assumes

```

    inv: cdclW-all-struct-inv b
    cdclW-merge++ b a
  shows (λS T. cdclW-all-struct-inv S ∧ 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 tranclp-into-rtranclp[OF st] cdclW-merge-rtranclp-cdclW
    assms(1) rtranclp-cdclW-all-struct-inv-inv rtranclp-mono[of cdclW-merge cdclW**] by fastforce
  then have (λS T. cdclW-all-struct-inv S ∧ cdclW-merge S T)++ c d
    using fw by auto
  then show ?case using IH by auto
qed

lemma backtrack-is-full1-cdclW-bj:
  assumes bt: backtrack S T and inv: cdclW-M-level-inv S
  shows full1 cdclW-bj S T
proof -
  have no-step cdclW-bj T
    using bt inv backtrack-no-cdclW-bj by blast
  moreover have cdclW-bj++ S T
    using bt by (auto intro: cdclW-bj.intros cdclW-o.intros cdclW.intros)
  ultimately show ?thesis unfolding full1-def by blast
qed

lemma rtrancl-cdclW-conflicting-true-cdclW-merge-restart:
  assumes cdclW** S V and inv: cdclW-M-level-inv S and conflicting S = None
  shows (cdclW-merge-restart** S V ∧ conflicting V = None)
    ∨ (∃ T U. cdclW-merge-restart** S T ∧ conflicting V ≠ None ∧ conflict T U ∧ cdclW-bj** U V)
  using assms
proof induction
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and cdclW = this(2) and IH = this(3)[OF this(4-)] and
    confl[simp] = this(5) and inv = this(4)
  from cdclW
  show ?case
  proof (cases)
    case propagate
    moreover then have conflicting U = None and conflicting V = None
      by (auto elim: propagateE)
    ultimately show ?thesis using IH cdclW-merge-restart.fw-r-propagate[of U V] by auto
  next
    case conflict
    moreover then have conflicting U = None and conflicting V ≠ None
      by (auto elim!: conflictE simp del: state-simp simp: state-eq-def)
    ultimately show ?thesis using IH by auto
  next
    case other
  then show ?thesis
  proof cases

```

```

    case decide
    then show ?thesis using IH cdclW-merge-restart.fw-r-decide[of U V] by (auto elim: decideE)
next
case bj
moreover {
  assume skip-or-resolve U V
  have f1: cdclW-bj++ U V
  by (simp add: local.bj tranclp.r-into-trancl)
  obtain T T' :: 'st where
    f2: cdclW-merge-restart** S U
    ∨ cdclW-merge-restart** S T ∧ conflicting U ≠ None
    ∧ conflict T T' ∧ cdclW-bj** T' U
  using IH confl by blast
  then have ?thesis
  proof -
    have conflicting V ≠ None ∧ conflicting U ≠ None
    using ⟨skip-or-resolve U V⟩
    by (auto simp: skip-or-resolve.simps state-eq-def elim!: skipE resolveE
        simp del: state-simp)
    then show ?thesis
    by (metis (no-types) IH f1 rtranclp-trans tranclp-into-rtranclp)
  qed
}
moreover {
  assume backtrack U V
  then have conflicting U ≠ None by (auto elim: backtrackE)
  then obtain T T' where
    cdclW-merge-restart** S T and
    conflicting U ≠ None and
    conflict T T' and
    cdclW-bj** T' U
  using IH confl by meson
  have invU: cdclW-M-level-inv U
  using inv rtranclp-cdclW-consistent-inv step.hyps(1) by blast
  then have conflicting V = None
  using ⟨backtrack U V⟩ inv by (auto elim: backtrack-levE
      simp: cdclW-M-level-inv-decomp)
  have full cdclW-bj T' V
  apply (rule rtranclp-fullI[of cdclW-bj T' U V])
  using ⟨cdclW-bj** T' U⟩ apply fast
  using ⟨backtrack U V⟩ backtrack-is-full1-cdclW-bj invU unfolding full1-def full-def
  by blast
  then have ?thesis
  using cdclW-merge-restart.fw-r-conflict[of T T' V] ⟨conflict T T'⟩
  ⟨cdclW-merge-restart** S T⟩ ⟨conflicting V = None⟩ by auto
}
ultimately show ?thesis by (auto simp: cdclW-bj.simps)
qed
next
case rf
moreover then have conflicting U = None and conflicting V = None
  by (auto simp: cdclW-rf.simps elim: restartE forgetE)
ultimately show ?thesis using IH cdclW-merge-restart.fw-r-rf[of U V] by auto
qed
qed

```


lemma *no-step-cdcl_W-no-step-cdcl_W-merge-restart*: *no-step cdcl_W S* \implies *no-step cdcl_W-merge-restart S*

by (auto simp: *cdcl_W.simps cdcl_W-merge-restart.simps cdcl_W-o.simps cdcl_W-bj.simps*)

lemma *no-step-cdcl_W-merge-restart-no-step-cdcl_W*:

assumes

conflicting S = None and

cdcl_W-M-level-inv S and

no-step cdcl_W-merge-restart S

shows *no-step cdcl_W S*

proof –

{ fix *S'*

assume *conflict S S'*

then have *cdcl_W S S'* using *cdcl_W.conflict* by auto

then have *cdcl_W-M-level-inv S'*

using *assms(2) cdcl_W-consistent-inv* by blast

then obtain *S''* where *full cdcl_W-bj S' S''*

using *cdcl_W-bj-exists-normal-form[of S']* by auto

then have *False*

using $\langle \text{conflict } S \ S' \rangle$ *assms(3) fw-r-conflict* by blast

}

then show *?thesis*

using *assms unfolding cdcl_W.simps cdcl_W-merge-restart.simps cdcl_W-o.simps cdcl_W-bj.simps*

by (auto elim: *skipE resolveE backtrackE conflictE decideE restartE*)

qed

lemma *cdcl_W-merge-restart-no-step-cdcl_W-bj*:

assumes

cdcl_W-merge-restart S T

shows *no-step cdcl_W-bj T*

using *assms*

by (*induction rule: cdcl_W-merge-restart.induct*)

(*force simp: cdcl_W-bj.simps cdcl_W-rf.simps cdcl_W-merge-restart.simps full-def*
elim!: rulesE)+

lemma *rtrancpl-cdcl_W-merge-restart-no-step-cdcl_W-bj*:

assumes

*cdcl_W-merge-restart** S T* and

conflicting S = None

shows *no-step cdcl_W-bj T*

using *assms unfolding rtrancpl-unfold*

apply (*elim disjE*)

apply (*force simp: cdcl_W-bj.simps cdcl_W-rf.simps elim!: rulesE*)

by (auto simp: *trancpl-unfold-end simp: cdcl_W-merge-restart-no-step-cdcl_W-bj*)

If *conflicting S* \neq *None*, we cannot say anything.

Remark that this theorem does not say anything about well-foundedness: even if you know that one relation is well-founded, it only states that the normal forms are shared.

lemma *conflicting-true-full-cdcl_W-iff-full-cdcl_W-merge*:

assumes *conf!*: *conflicting S = None* and *lev: cdcl_W-M-level-inv S*

shows *full cdcl_W S V* \longleftrightarrow *full cdcl_W-merge-restart S V*

proof

assume *full: full cdcl_W-merge-restart S V*

then have *st: cdcl_W** S V*

```

using rtrancp-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 rtrancp-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 rtrancp-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) cdclW-merge-restart** S V and conflicting V = None
  | (bj) T U where
    cdclW-merge-restart** S T and
    conflicting V  $\neq$  None and
    conflict T U and
    cdclW-bj** U V
  using full rtrancp-cdclW-conflicting-true-cdclW-merge-restart confl lev unfolding full-def
  by meson
then have cdclW-merge-restart** S V
proof cases
  case fw
    then show ?thesis by fast
  next
    case (bj T U)
      have no-step cdclW-bj V
        using full unfolding full-def by (meson cdclW-o.bj other)
      then have full cdclW-bj U V
        using  $\langle \text{cdcl}_W\text{-bj}^{**} U V \rangle$  unfolding full-def by auto
      then have cdclW-merge-restart T V
        using  $\langle \text{conflict } T U \rangle$  cdclW-merge-restart.fw-r-conflict by blast
      then show ?thesis using  $\langle \text{cdcl}_W\text{-merge-restart}^{**} S T \rangle$  by auto
    qed
ultimately show full cdclW-merge-restart S V unfolding full-def by fast
qed

lemma init-state-true-full-cdclW-iff-full-cdclW-merge:
  shows full cdclW (init-state N) V  $\longleftrightarrow$  full cdclW-merge-restart (init-state N) V
  by (rule conflicting-true-full-cdclW-iff-full-cdclW-merge) auto

```

20.4 FW with strategy

20.4.1 The intermediate step

inductive cdcl_W-s' :: 'st \Rightarrow 'st \Rightarrow bool **where**
 conflict': full1 cdcl_W-cp S S' \implies cdcl_W-s' S S' |
 decide': decide S S' \implies no-step cdcl_W-cp S \implies full cdcl_W-cp S' S'' \implies cdcl_W-s' S S'' |

$bj': full1\ cdcl_W-bj\ S\ S' \implies no-step\ cdcl_W-cp\ S \implies full\ cdcl_W-cp\ S'\ S'' \implies cdcl_W-s'\ S\ S''$

inductive-cases $cdcl_W-s'E: cdcl_W-s'\ S\ T$

lemma $rtrancp-cdcl_W-bj-full1-cdclp-cdcl_W-stgy:$

$cdcl_W-bj^{**}\ S\ S' \implies full\ cdcl_W-cp\ S'\ S'' \implies cdcl_W-stgy^{**}\ S\ S''$

proof (induction rule: converse-rtrancp-induct)

case *base*

then show $?case$ **by** (metis $cdcl_W-stgy.conflict'$ full-unfold $rtrancp.simps$)

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: $cdcl_W-bj.simps\ cdcl_W-cp.simps\ elim!$: rulesE)

consider

(U) $U = S'$

| (U') U' **where** $cdcl_W-bj\ U\ U'$ **and** $cdcl_W-bj^{**}\ U'\ S'$

using st **by** (metis converse-rtrancpE)

then show $?case$

proof *cases*

case U

then show $?thesis$

using ($no-step\ cdcl_W-cp\ T$) $cdcl_W-o.bj\ local.bj\ other'$ step.prem **by** (meson $r-into-rtrancp$)

next

case U' **note** $U' = this(1)$

have $no-step\ cdcl_W-cp\ U$

using U' **by** (fastforce simp: $cdcl_W-cp.simps\ cdcl_W-bj.simps\ elim!$: rulesE)

then have $full\ cdcl_W-cp\ U\ U$

by (simp add: full-unfold)

then have $cdcl_W-stgy\ T\ U$

using ($no-step\ cdcl_W-cp\ T$) $cdcl_W-stgy.simps\ local.bj\ cdcl_W-o.bj$ **by** meson

then show $?thesis$ **using** IH **by** auto

qed

qed

lemma $cdcl_W-s'-is-rtrancp-cdcl_W-stgy:$

$cdcl_W-s'\ S\ T \implies cdcl_W-stgy^{**}\ S\ T$

apply (induction rule: $cdcl_W-s'.induct$)

apply (auto intro: $cdcl_W-stgy.intros$)[]

apply (meson decide $other'\ r-into-rtrancp$)

by (metis full1-def $rtrancp-cdcl_W-bj-full1-cdclp-cdcl_W-stgy\ trancp-into-rtrancp$)

lemma $cdcl_W-cp-cdcl_W-bj-bissimulation:$

assumes

$full\ cdcl_W-cp\ T\ U$ **and**

$cdcl_W-bj^{**}\ T\ T'$ **and**

$cdcl_W-all-struct-inv\ T$ **and**

$no-step\ cdcl_W-bj\ T'$

shows $full\ cdcl_W-cp\ T'\ U$

$\vee (\exists U'\ U'', full\ cdcl_W-cp\ T'\ U'' \wedge full1\ cdcl_W-bj\ U\ U' \wedge full\ cdcl_W-cp\ U'\ U''$
 $\wedge cdcl_W-s'^{**}\ U\ U'')$

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  $cdcl_W\text{-}bj^{**}\ T\ T''$ 
  using  $local.bj\ st$  by  $auto$ 
then have  $cdcl_W^{**}\ T\ T''$ 
  using  $rtrancp\text{-}cdcl_W\text{-}bj\text{-}rtrancp\text{-}cdcl_W$  by  $blast$ 
then have  $inv\text{-}T'':\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ T''$ 
  using  $inv\ rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$  by  $blast$ 
have  $cdcl_W\text{-}bj^{++}\ T\ T''$ 
  using  $local.bj\ st$  by  $auto$ 
have  $full1\ cdcl_W\text{-}bj\ T\ T''$ 
  by ( $metis\ \langle cdcl_W\text{-}bj^{++}\ T\ T'' \rangle\ full1\text{-}def\ step.prem(3)$ )
then have  $T = U$ 
proof –
  obtain  $Z$  where  $cdcl_W\text{-}bj\ T\ Z$ 
    using  $\langle cdcl_W\text{-}bj^{++}\ T\ T'' \rangle$  by ( $blast\ dest:\ trancpD$ )
  { assume  $cdcl_W\text{-}cp^{++}\ T\ U$ 
    then obtain  $Z'$  where  $cdcl_W\text{-}cp\ T\ Z'$ 
      by ( $meson\ trancpD$ )
    then have  $False$ 
      using  $\langle cdcl_W\text{-}bj\ T\ Z \rangle$  by ( $fastforce\ simp:\ cdcl_W\text{-}bj.simps\ cdcl_W\text{-}cp.simps$ 
         $elim:\ rulesE$ )
    }
  then show  $?thesis$ 
    using  $full\ unfolding\ full\text{-}def\ rtrancp\text{-}unfold$  by  $blast$ 
qed
obtain  $U''$  where  $full\ cdcl_W\text{-}cp\ T''\ U''$ 
  using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv\ inv\text{-}T''$  by  $blast$ 
moreover then have  $cdcl_W\text{-}stgy^{**}\ U\ U''$ 
  by ( $metis\ \langle T = U \rangle\ \langle cdcl_W\text{-}bj^{++}\ T\ T'' \rangle\ rtrancp\text{-}cdcl_W\text{-}bj\text{-}full1\text{-}cdclp\text{-}cdcl_W\text{-}stgy\ rtrancp\text{-}unfold$ )
moreover have  $cdcl_W\text{-}s^{**}\ U\ U''$ 
proof –
  obtain  $ss :: 'st \Rightarrow 'st$  where
     $f1: \forall x2. (\exists v3. cdcl_W\text{-}cp\ x2\ v3) = cdcl_W\text{-}cp\ x2\ (ss\ x2)$ 
    by  $moura$ 
  have  $\neg cdcl_W\text{-}cp\ U\ (ss\ U)$ 
    by ( $meson\ full\ full\text{-}def$ )
  then show  $?thesis$ 
    using  $f1$  by ( $metis\ (no\text{-}types)\ \langle T = U \rangle\ \langle full1\ cdcl_W\text{-}bj\ T\ T'' \rangle\ bj'\ calculation(1)$ 
       $r\text{-}into\text{-}rtrancp$ )
qed
ultimately show  $?case$ 
  using  $\langle full1\ cdcl_W\text{-}bj\ T\ T'' \rangle\ \langle full\ cdcl_W\text{-}cp\ T''\ U'' \rangle\ unfolding\ \langle T = U \rangle$  by  $blast$ 
qed

```

lemma $cdcl_W\text{-}cp\text{-}cdcl_W\text{-}bj\text{-}bissimulation'$:

assumes

$full\ cdcl_W\text{-}cp\ T\ U$ **and**

$cdcl_W\text{-}bj^{**}\ T\ T'$ **and**

$cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$ **and**

$no\text{-}step\ cdcl_W\text{-}bj\ T'$

shows $full\ cdcl_W\text{-}cp\ T'\ U$

$\vee (\exists U'. full1\ cdcl_W\text{-}bj\ U\ U' \wedge (\forall U''. full\ cdcl_W\text{-}cp\ U'\ U'' \longrightarrow full\ cdcl_W\text{-}cp\ T'\ U''$
 $\wedge cdcl_W\text{-}s^{**}\ U\ U''))$

using $assms(2,1,3,4)$

```

proof (induction rule: rtrancpl-induct)
  case base
  then show ?case by blast
next
  case (step  $T'$   $T''$ ) note  $st = \text{this}(1)$  and  $bj = \text{this}(2)$  and  $IH = \text{this}(3)[OF \text{this}(4,5)]$  and
     $full = \text{this}(4)$  and  $inv = \text{this}(5)$ 
  have  $cdcl_W^{**} T T''$ 
    by (metis local.bj rtrancpl.simps rtrancpl-cdclW-bj-rtrancpl-cdclW st)
  then have  $inv\text{-}T''$ :  $cdcl_W\text{-all-struct-inv } T''$ 
    using  $inv$  rtrancpl-cdclW-all-struct-inv-inv by blast
  have  $cdcl_W\text{-bj}^{++} T T''$ 
    using local.bj st by auto
  have full1  $cdcl_W\text{-bj } T T''$ 
    by (metis  $\langle cdcl_W\text{-bj}^{++} T T'' \rangle$  full1-def step.premis(3))
  then have  $T = U$ 
  proof –
    obtain  $Z$  where  $cdcl_W\text{-bj } T Z$ 
      using  $\langle cdcl_W\text{-bj}^{++} T T'' \rangle$  by (blast dest: trancplD)
    { assume  $cdcl_W\text{-cp}^{++} T U$ 
      then obtain  $Z'$  where  $cdcl_W\text{-cp } T Z'$ 
        by (meson trancplD)
      then have False
        using  $\langle cdcl_W\text{-bj } T Z \rangle$  by (fastforce simp:  $cdcl_W\text{-bj.simps } cdcl_W\text{-cp.simps elim: rulesE$ )
    }
    then show ?thesis
      using full unfolding full-def rtrancpl-unfold by blast
  qed
{ fix  $U''$ 
  assume full  $cdcl_W\text{-cp } T'' U''$ 
  moreover then have  $cdcl_W\text{-stgy}^{**} U U''$ 
    by (metis  $\langle T = U \rangle \langle cdcl_W\text{-bj}^{++} T T'' \rangle$  rtrancpl-cdclW-bj-full1-cdclW-stgy rtrancpl-unfold)
  moreover have  $cdcl_W\text{-s}^{**} U U''$ 
  proof –
    obtain  $ss :: 'st \Rightarrow 'st$  where
       $f1: \forall x2. (\exists v3. cdcl_W\text{-cp } x2 v3) = cdcl_W\text{-cp } x2 (ss x2)$ 
    by moura
    have  $\neg cdcl_W\text{-cp } U (ss U)$ 
      by (meson assms(1) full-def)
    then show ?thesis
      using f1 by (metis (no-types)  $\langle T = U \rangle \langle full1 cdcl_W\text{-bj } T T'' \rangle$  bj' calculation(1)
        r-into-rtrancpl)
  qed
  ultimately have full1  $cdcl_W\text{-bj } U T''$  and  $cdcl_W\text{-s}^{**} T'' U''$ 
    using  $\langle full1 cdcl_W\text{-bj } T T'' \rangle \langle full cdcl_W\text{-cp } T'' U'' \rangle$  unfolding  $\langle T = U \rangle$ 
    apply blast
    by (metis  $\langle full cdcl_W\text{-cp } T'' U'' \rangle cdcl_W\text{-s'.simps full-unfold rtrancpl.simps}$ )
  }
then show ?case
  using  $\langle full1 cdcl_W\text{-bj } T T'' \rangle$  full bj' unfolding  $\langle T = U \rangle$  full-def by (metis r-into-rtrancpl)
qed

lemma  $cdcl_W\text{-stgy-cdcl}_W\text{-s}'\text{-connected}$ :
  assumes  $cdcl_W\text{-stgy } S U$  and  $cdcl_W\text{-all-struct-inv } S$ 
  shows  $cdcl_W\text{-s}' S U$ 
     $\vee (\exists U'. full1 cdcl_W\text{-bj } U U' \wedge (\forall U''. full cdcl_W\text{-cp } U' U'' \longrightarrow cdcl_W\text{-s}' S U''))$ 

```

```

using assms
proof (induction rule: 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 inv-T: cdclW-all-struct-inv T
    using cdclW-all-struct-inv-inv o other other'.prems by blast
  consider
    (cp) full cdclW-cp T U and no-step cdclW-bj T
  | (fbj) T' where full1 cdclW-bj T T'
  apply (cases no-step cdclW-bj T)
    using full apply blast
  using cdclW-bj-exists-normal-form[of T] inv-T unfolding cdclW-all-struct-inv-def
  by (metis full-unfold)
  then show ?thesis
  proof cases
    case cp
    then show ?thesis
    proof -
      obtain ss :: 'st ⇒ 'st where
        f1: ∀ s sa sb. (¬ full1 cdclW-bj s sa ∨ cdclW-cp s (ss s) ∨ ¬ full cdclW-cp sa sb)
          ∨ cdclW-s' s sb
      using bj' by moura
      have full1 cdclW-bj S T
        by (simp add: cp(2) full1-def local.bj tranclp.r-into-trancl)
      then show ?thesis
        using f1 full n-s by blast
    qed
  next
  case (fbj U')
  then have full1 cdclW-bj S U'
    using bj unfolding full1-def by auto
  moreover have no-step cdclW-cp S
    using n-s by blast
  moreover have T = U
    using full fbj unfolding full1-def full-def rtranclp-unfold
    by (force dest!: tranclpD simp: cdclW-bj.simps elim: rulesE)
  ultimately show ?thesis using cdclW-s'.bj'[of S U] using fbj by blast
  qed
qed
qed
qed

```

lemma *cdcl_W-stgy-cdcl_W-s'-connected'*:
 assumes *cdcl_W-stgy S U* and *cdcl_W-all-struct-inv S*

shows $cdcl_W-s' S U$
 $\vee (\exists U' U''. cdcl_W-s' S U'' \wedge full1\ cdcl_W-bj\ U\ U' \wedge full\ cdcl_W-cp\ U'\ U'')$
using *assms*
proof (*induction rule: cdcl_W-stgy.induct*)
case (*conflict'* T)
then have $cdcl_W-s' S T$
using $cdcl_W-s'.conflict'$ **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** $cdcl_W-s'.simps\ full\ n-s$ **by** *blast*
next
case *bj*
have $cdcl_W-all-struct-inv\ T$
using $cdcl_W-all-struct-inv-inv\ o\ other\ other'.prems$ **by** *blast*
then obtain T' **where** $T': full\ cdcl_W-bj\ T\ T'$
using $cdcl_W-bj-exists-normal-form$ **unfolding** $full-def\ cdcl_W-all-struct-inv-def$ **by** *metis*
then have $full\ cdcl_W-bj\ S\ T'$
proof –
have $f1: cdcl_W-bj^{**}\ T\ T' \wedge no-step\ cdcl_W-bj\ T'$
by (*metis* (*no-types*) $T'\ full-def$)
then have $cdcl_W-bj^{**}\ S\ T'$
by (*meson converse-rtranclp-into-rtranclp local.bj*)
then show *?thesis*
using $f1$ **by** (*simp add: full-def*)
qed
have $cdcl_W-bj^{**}\ T\ T'$
using T' **unfolding** $full-def$ **by** *simp*
have $cdcl_W-all-struct-inv\ T$
using $cdcl_W-all-struct-inv-inv\ o\ other\ other'.prems$ **by** *blast*
then consider
 $(T'U)\ full\ cdcl_W-cp\ T'\ U$
 $| (U)\ U'\ U''$ **where**
 $full\ cdcl_W-cp\ T'\ U''$ **and**
 $full1\ cdcl_W-bj\ U\ U'$ **and**
 $full\ cdcl_W-cp\ U'\ U''$ **and**
 $cdcl_W-s'^{**}\ U\ U''$
using $cdcl_W-cp-cdcl_W-bj-bissimulation[OF\ full\ \langle cdcl_W-bj^{**}\ T\ T' \rangle]$ T' **unfolding** $full-def$
by *blast*
then show *?thesis* **by** (*metis* $T'\ cdcl_W-s'.simps\ full-full1\ local.bj\ n-s$)
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  $cdcl_W\text{-stgy}^{**} S U$  and  $inv: cdcl_W\text{-M-level-inv } S$ 
shows  $cdcl_W\text{-s}'^{**} S U \vee (\exists T. cdcl_W\text{-s}'^{**} S T \wedge cdcl_W\text{-bj}^{++} T U \wedge conflicting\ U \neq None)$ 
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  $f2: cdcl_W\text{-s}'\ T\ V$ 
    using  $cdcl_W\text{-s}'.conflict'$  by blast
    obtain  $ss :: 'st$  where
       $f3: S = T \vee cdcl_W\text{-stgy}^{**} S ss \wedge cdcl_W\text{-stgy}\ ss\ T$ 
      by ( $metis\ (full\text{-types})\ rtranclp.simps\ st$ )
    obtain  $ssa :: 'st$  where
       $ssa: cdcl_W\text{-cp}\ T\ ssa$ 
      using conflict' by ( $metis\ (no\text{-types})\ full1\text{-def}\ tranclpD$ )
    have  $\forall s. \neg full\ cdcl_W\text{-cp}\ s\ T$ 
    by ( $meson\ ssa\ full\text{-def}$ )
    then have  $S = T$ 
    by ( $metis\ (full\text{-types})\ f3\ ssa\ cdcl_W\text{-stgy.cases}\ full1\text{-def}$ )
    then show  $?thesis$ 
    using  $f2$  by blast
  next
  case ( $other'\ U$ ) note  $o = this(1)$  and  $n\text{-s} = this(2)$  and  $full = this(3)$ 
  then show  $?thesis$ 
  using  $o$ 
  proof ( $cases\ rule: cdcl_W\text{-o-rule-cases}$ )
    case decide
    then have  $cdcl_W\text{-s}'^{**} S\ T$ 
    using  $IH$  by ( $auto\ elim: rulesE$ )
    then show  $?thesis$ 
    by ( $meson\ decide\ decide'\ full\ n\text{-s}\ rtranclp.rtrancl\text{-into-rtrancl}$ )
  next
  case backtrack
  consider
    ( $s'$ )  $cdcl_W\text{-s}'^{**} S\ T$ 
    | ( $bj$ )  $S'$  where  $cdcl_W\text{-s}'^{**} S\ S'$  and  $cdcl_W\text{-bj}^{++} S'\ T$  and  $conflicting\ T \neq None$ 
    using  $IH$  by blast
  then show  $?thesis$ 
  proof cases
    case  $s'$ 
    moreover
    have  $cdcl_W\text{-M-level-inv}\ T$ 
    using  $inv\ local.step(1)\ rtranclp\text{-}cdcl_W\text{-stgy-consistent-inv}$  by auto
    then have  $full1\ cdcl_W\text{-bj}\ T\ U$ 
    using backtrack-is-full1-cdcl_W-bj backtrack by blast
    then have  $cdcl_W\text{-s}'\ T\ V$ 
    using  $full\ bj'\ n\text{-s}$  by blast
    ultimately show  $?thesis$  by auto
  next
  case ( $bj\ S'$ ) note  $S\text{-}S' = this(1)$  and  $bj\text{-}T = this(2)$ 
  have  $no\text{-step}\ cdcl_W\text{-cp}\ S'$ 

```



```

    using bj-T by (fastforce simp: cdclW-cp.simps cdclW-bj.simps dest!: tranclpD
      elim: rulesE)
  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 elim: rulesE)
then have confl-V: conflicting V ≠ None
  using skip by (auto elim!: rulesE simp del: state-simp simp: state-eq-def)
consider
  (s') cdclW-s'^** S T
  | (bj) S' where cdclW-s'^** S S' and cdclW-bj++ S' T and conflicting T ≠ None
  using IH by blast
then show ?thesis
proof cases
case s'
show ?thesis using s' confl-V skip by force
next
case (bj S') note S-S' = this(1) and bj-T = this(2)
have cdclW-bj++ S' V
  using skip bj-T by (metis ⟨U = V⟩ cdclW-bj.skip tranclp.simps)
then show ?thesis using S-S' confl-V by auto
qed
next
case resolve
then have [simp]: U = V
  using full unfolding full-def rtranclp-unfold
  by (auto elim!: rulesE dest!: tranclpD
    simp del: state-simp simp: state-eq-def cdclW-cp.simps)
have confl-V: conflicting V ≠ None
  using resolve by (auto elim!: rulesE simp del: state-simp simp: state-eq-def)

consider
  (s') cdclW-s'^** S T
  | (bj) S' where cdclW-s'^** S S' and cdclW-bj++ S' T and conflicting T ≠ None
  using IH by blast
then show ?thesis
proof cases
case s'
have cdclW-bj++ T V
  using resolve by force
then show ?thesis using s' confl-V by auto
next
case (bj S') note S-S' = this(1) and bj-T = this(2)
have cdclW-bj++ S' V
  using resolve bj-T by (metis ⟨U = V⟩ cdclW-bj.resolve tranclp.simps)

```

```

      then show ?thesis using confl-V S-S' by auto
    qed
  qed
  qed
  qed

lemma n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o:
  assumes inv: cdclW-all-struct-inv S
  shows no-step cdclW-s' S  $\longleftrightarrow$  no-step cdclW-cp S  $\wedge$  no-step cdclW-o S (is ?S' S  $\longleftrightarrow$  ?C S  $\wedge$  ?O S)
proof
  assume ?C S  $\wedge$  ?O S
  then show ?S' S
    by (auto simp: cdclW-s'.simps full1-def tranclp-unfold-begin)
next
  assume n-s: ?S' S
  have ?C S
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then obtain S' where cdclW-cp S S'
      by auto
    then obtain T where full1 cdclW-cp S T
      using cdclW-cp-normalized-element-all-inv inv by (metis (no-types, lifting) full-unfold)
    then show False using n-s cdclW-s'.conflict' by blast
  qed
  moreover have ?O S
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then obtain S' where cdclW-o S S'
      by auto
    then obtain T where full1 cdclW-cp S' T
      using cdclW-cp-normalized-element-all-inv inv
      by (meson cdclW-all-struct-inv-def n-s
        cdclW-stgy-cdclW-s'-connected' cdclW-then-exists-cdclW-stgy-step )
    then show False using n-s by (meson (cdclW-o S S') cdclW-all-struct-inv-def
      cdclW-stgy-cdclW-s'-connected' cdclW-then-exists-cdclW-stgy-step inv)
  qed
  ultimately show ?C S  $\wedge$  ?O S by auto
qed

lemma cdclW-s'-tranclp-cdclW:
  cdclW-s' S S'  $\implies$  cdclW++ S S'
proof (induct rule: cdclW-s'.induct)
  case conflict'
  then show ?case
    by (simp add: full1-def tranclp-cdclW-cp-tranclp-cdclW)
next
  case decide'
  then show ?case
    using cdclW-stgy.simps cdclW-stgy-tranclp-cdclW by (meson cdclW-o.simps)
next
  case (bj' Sa S'a S'') note a2 = this(1) and a1 = this(2) and n-s = this(3)
  obtain ss :: 'st  $\Rightarrow$  '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) tranclpD)
 have $cdcl_W\text{-}bj^{++} Sa S'a \wedge \text{no-step } cdcl_W\text{-}bj S'a$
 using a2 by (simp add: full1-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-s by (meson bj other rtranclp-cdcl_W\text{-}bj-full1-cdclp-cdcl_W\text{-}stgy
 rtranclp-cdcl_W\text{-}stgy-rtranclp-cdcl_W rtranclp-into-tranclp2)
 qed

lemma $tranclp\text{-}cdcl_W\text{-}s'\text{-}tranclp\text{-}cdcl_W$:
 $cdcl_W\text{-}s'^{++} S S' \implies cdcl_W^{++} S S'$
apply (induct rule: tranclp.induct)
using $cdcl_W\text{-}s'\text{-}tranclp\text{-}cdcl_W$ **apply** blast
by (meson $cdcl_W\text{-}s'\text{-}tranclp\text{-}cdcl_W$ tranclp-trans)

lemma $rtranclp\text{-}cdcl_W\text{-}s'\text{-}rtranclp\text{-}cdcl_W$:
 $cdcl_W\text{-}s'^{**} S S' \implies cdcl_W^{**} S S'$
using $rtranclp\text{-}unfold[of\ cdcl_W\text{-}s' S S']\ tranclp\text{-}cdcl_W\text{-}s'\text{-}tranclp\text{-}cdcl_W[of\ S S']$ **by** auto

lemma $full\text{-}cdcl_W\text{-}stgy\text{-}iff\text{-}full\text{-}cdcl_W\text{-}s'$:
assumes inv : $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $full\ cdcl_W\text{-}stgy\ S\ T \longleftrightarrow full\ cdcl_W\text{-}s'\ S\ T$ (is $?S \longleftrightarrow ?S'$)

proof

assume $?S'$
then have $cdcl_W^{**} S\ T$
using $rtranclp\text{-}cdcl_W\text{-}s'\text{-}rtranclp\text{-}cdcl_W[of\ S\ T]$ **unfolding** $full\text{-}def$ **by** blast
then have inv' : $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$
using $rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv\ inv$ **by** blast
have $cdcl_W\text{-}stgy^{**} S\ T$
using $\langle ?S' \rangle$ **unfolding** $full\text{-}def$
using $cdcl_W\text{-}s'\text{-}is\text{-}rtranclp\text{-}cdcl_W\text{-}stgy\ rtranclp\text{-}mono[of\ cdcl_W\text{-}s'\ cdcl_W\text{-}stgy^{**}]$ **by** auto
then show $?S$
using $\langle ?S' \rangle\ inv'\ cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}s'\text{-}connected'$ **unfolding** $full\text{-}def$ **by** blast

next

assume $?S$
then have $inv\text{-}T$: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$
by (metis $assms\ full\text{-}def\ rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv\ rtranclp\text{-}cdcl_W\text{-}stgy\text{-}rtranclp\text{-}cdcl_W$)

consider

$(s')\ cdcl_W\text{-}s'^{**} S\ T$
 $| (st)\ S'$ **where** $cdcl_W\text{-}s'^{**} S\ S'$ **and** $cdcl_W\text{-}bj^{++} S'\ T$ **and** $conflicting\ T \neq None$
using $rtranclp\text{-}cdcl_W\text{-}stgy\text{-}connected\text{-}to\text{-}rtranclp\text{-}cdcl_W\text{-}s'[of\ S\ T]\ inv\ \langle ?S \rangle$
unfolding $full\text{-}def\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$
by blast

then show $?S'$

proof cases

case s'

then show $?thesis$

by (metis $\langle full\ cdcl_W\text{-}stgy\ S\ T \rangle\ inv\text{-}T\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}s'\text{-}simps$
 $cdcl_W\text{-}stgy\text{-}conflict'\ cdcl_W\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step\ full\text{-}def$
 $n\text{-step}\text{-}cdcl_W\text{-}stgy\text{-}iff\text{-}no\text{-}step\text{-}cdcl_W\text{-}cl\text{-}cdcl_W\text{-}o$)

next

case $(st\ S')$

have $full\ cdcl_W\text{-}cp\ T\ T$

```

    using option-full-cdclW-cp st(3) by blast
  moreover
    have n-s: no-step cdclW-bj T
      by (metis ⟨full cdclW-stgy S T⟩ bj inv-T cdclW-all-struct-inv-def
        cdclW-then-exists-cdclW-stgy-step full-def)
    then have full1 cdclW-bj S' T
      using st(2) unfolding full1-def by blast
  moreover have no-step cdclW-cp S'
    using st(2) by (fastforce dest!: tranclpD simp: cdclW-cp.simps cdclW-bj.simps
      elim: rulesE)
  ultimately have cdclW-s' S' T
    using cdclW-s'.bj'[of S' T T] by blast
  then have cdclW-sfs* 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 ∃ T. cdclW-stgy S T
proof -
  obtain U where full cdclW-cp S U
    using cdclW-cp-normalized-element-all-inv assms by blast
  then have full1 cdclW-cp S U
    by (metis cdclW-cp.conflict' assms(1) full-unfold)
  then show ?thesis using cdclW-stgy.conflict' by blast
qed

```

```

lemma decide-step-cdclW-stgy-step:
  assumes
    decide S T
    cdclW-all-struct-inv S
  shows ∃ T. cdclW-stgy S T
proof -
  obtain U where full cdclW-cp T U
    using cdclW-cp-normalized-element-all-inv by (meson assms(1) assms(2) cdclW-all-struct-inv-inv
      cdclW-cp-normalized-element-all-inv decide other)
  then show ?thesis
    by (metis assms cdclW-cp-normalized-element-all-inv cdclW-stgy.conflict' decide full-unfold
      other')
qed

```

```

lemma rtranclp-cdclW-cp-conflicting-Some:
  cdclW-cpfs* S T ⟹ conflicting S = Some D ⟹ S = T
  using rtranclpD tranclpD by fastforce

```

```

inductive cdclW-merge-cp :: 'st ⇒ 'st ⇒ bool where
  conflict'[intro]: conflict S T ⟹ full cdclW-bj T U ⟹ cdclW-merge-cp S U |

```

propagate'[intro]: $\text{propagate}^{++} S S' \implies \text{cdcl}_W\text{-merge-cp } S S'$

lemma *cdcl_W-merge-restart-cases*[consumes 1, case-names conflict propagate]:

assumes

cdcl_W-merge-cp $S U$ **and**

$\bigwedge T. \text{conflict } S T \implies \text{full } \text{cdcl}_W\text{-bj } T U \implies P$ **and**

$\text{propagate}^{++} S U \implies P$

shows P

using *assms unfolding cdcl_W-merge-cp.simps* **by** *auto*

lemma *cdcl_W-merge-cp-tranclp-cdcl_W-merge*:

cdcl_W-merge-cp $S T \implies \text{cdcl}_W\text{-merge}^{++} S T$

apply (*induction rule: cdcl_W-merge-cp.induct*)

using *cdcl_W-merge.simps apply auto*[1]

using *tranclp-mono*[of *propagate cdcl_W-merge*] *fw-propagate* **by** *blast*

lemma *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W*:

cdcl_W-merge-cp^{**} $S T \implies \text{cdcl}_W$ ^{**} $S T$

apply (*induction rule: rtranclp-induct*)

apply *simp*

unfolding *cdcl_W-merge-cp.simps* **by** (*meson cdcl_W-merge-restart-cdcl_W 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 $S T \implies \text{no-step } \text{cdcl}_W\text{-cp } S$

by (*metis rtranclp-unfold cdcl_W-cp-conflicting-not-empty option.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]: $\text{full1 } \text{cdcl}_W\text{-cp } S S' \implies \text{cdcl}_W\text{-s'-without-decide } S S' \mid$

bj'-without-decide[intro]: $\text{full1 } \text{cdcl}_W\text{-bj } S S' \implies \text{no-step } \text{cdcl}_W\text{-cp } S \implies \text{full } \text{cdcl}_W\text{-cp } S' S'' \implies \text{cdcl}_W\text{-s'-without-decide } S S''$

lemma *rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W*:

cdcl_W-s'-without-decide^{**} $S T \implies \text{cdcl}_W$ ^{**} $S T$

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^{**} $S T \implies \text{cdcl}_W\text{-s}'^{\text{'**}}$ $S T$

proof (*induction rule: rtranclp-induct*)

case *base*

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

next

case (*step* $y z$) **note** $a2 = \text{this}(2)$ **and** $a1 = \text{this}(3)$

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

```

    cdclW-merge-cp** S V
    conflicting S = None
shows
  (cdclW-s'-without-decide** S V)
  ∨ (∃ T. cdclW-s'-without-decide** S T ∧ propagate++ T V)
  ∨ (∃ T U. cdclW-s'-without-decide** S T ∧ full1 cdclW-bj T U ∧ propagate** U V)
using assms
proof (induction rule: rtrancp-induct)
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and cp = this(2) and IH = this(3)[OF this(4)]
  from cp show ?case
  proof (cases rule: cdclW-merge-restart-cases)
    case propagate
    then show ?thesis using IH by (meson rtrancp-trancp-trancp trancp-into-rtrancp)
  next
    case (conflict U') note confl = this(1) and bj = this(2)
    have full1-U-U': full1 cdclW-cp U U'
    by (simp add: conflict-is-full1-cdclW-cp local.conflict(1))
    consider
      (s') cdclW-s'-without-decide** S U
    | (propa) T' where cdclW-s'-without-decide** S T' and propagate++ T' U
    | (bj-prop) T' T'' where
      cdclW-s'-without-decide** S T' and
      full1 cdclW-bj T' T'' and
      propagate** T'' U
    using IH by blast
  then show ?thesis
  proof cases
    case s'
    have cdclW-s'-without-decide U U'
    using full1-U-U' conflict'-without-decide by blast
    then have cdclW-s'-without-decide** S U'
    using ⟨cdclW-s'-without-decide** S U⟩ by auto
    moreover have U' = V ∨ full1 cdclW-bj U' V
    using bj by (meson full-unfold)
    ultimately show ?thesis by blast
  next
    case propa note s' = this(1) and T'-U = this(2)
    have full1 cdclW-cp T' U'
    using rtrancp-mono[of propagate cdclW-cp] T'-U cdclW-cp.propagate' full1-U-U'
    rtrancp-full1I[of cdclW-cp T'] by (metis (full-types) predicate2D predicate2I
      trancp-into-rtrancp)
    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 rtrancp-mono[of propagate cdclW-cp] T''-U cdclW-cp.propagate' full1-U-U'
    rtrancp-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' rtrancp.intros(2)[of - S T' U'] by blast
  then show ?thesis
    by (metis full-unfold local.bj rtrancp.rtrancp-refl)
qed
qed
qed

lemma rtrancp-cdclW-s'-without-decide-is-rtrancp-cdclW-merge-cp:
  assumes
    cdclW-s'-without-decide** S V and
    confl: conflicting S = None
  shows
    (cdclW-merge-cp** S V ∧ conflicting V = None)
    ∨ (cdclW-merge-cp** S V ∧ conflicting V ≠ None ∧ no-step cdclW-cp V ∧ no-step cdclW-bj V)
    ∨ (∃ T. cdclW-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 = None
      using trancp-cdclW-cp-propagate-with-conflict-or-not[of U V]
      conflict by (auto dest!: trancpD simp: rtrancp-unfold elim: rulesE)
    then have cdclW-merge-cp** S U using IH by (auto elim: rulesE
      simp del: state-simp simp: state-eq-def)
    consider
      (propa) propagate++ U V
      | (confl') conflict U V
      | (propa-confl') U' where propagate++ U U' conflict U' V
    using trancp-cdclW-cp-propagate-with-conflict-or-not[OF rt] unfolding rtrancp-unfold
    by fastforce
  then show ?thesis
  proof cases
    case propa
    then have cdclW-merge-cp U V
      by auto
    moreover have conflicting V = None
      using propa unfolding trancp-unfold-end by (auto elim: rulesE)
    ultimately show ?thesis using ⟨cdclW-merge-cp** S U⟩ by (auto elim!: rulesE
      simp del: state-simp simp: state-eq-def)
  next
    case confl'
    then show ?thesis using ⟨cdclW-merge-cp** S U⟩ by auto
  next

```

```

    case propa-conf' 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 ≠ None
  using full-bj unfolding full1-def by (fastforce dest!: tranclpD simp: cdclW-bj.simps
    elim: rulesE)
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-conf') 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 cdclW-merge-cp U' V by auto
  moreover have conflicting V = None
    using propa unfolding tranclp-unfold-end by (auto elim: rulesE)
  ultimately show ?thesis using S-U' by (auto elim: rulesE
    simp del: state-simp simp: state-eq-def)
next
case confl'
  then show ?thesis using S-U' by auto
next
case propa-conf' note propa = this(1) and confl = this(2)
  have cdclW-merge-cp U' U'' using propa by auto
  then show ?thesis using S-U' confl by (meson rtranclp.rtrancl-into-rtrancl)
next
case n-s
  then show ?thesis
    using S-U' apply (cases conflicting V = None)
    using full-bj apply simp
    by (metis cp full-def full-unfold full-bj)
qed
qed
qed

```

lemma *no-step-cdcl_W-s'-no-ste-cdcl_W-merge-cp:*
assumes
 cdcl_W-all-struct-inv S
 conflicting S = None
 no-step cdcl_W-s' S
shows no-step cdcl_W-merge-cp S
using *assms* **apply** (auto simp: cdcl_W-s'.simps cdcl_W-merge-cp.simps)


```

    using conflict-is-full1-cdclW-cp apply blast
using cdclW-cp-normalized-element-all-inv cdclW-cp.propagate' by (metis cdclW-cp.propagate'
full-unfold tranclpD)

```

The *no-step decide* S is needed, since $cdcl_W$ -merge-cp is $cdcl_W$ -s' without *decide*.

lemma *conflicting-true-no-step-cdcl_W-merge-cp-no-step-s'-without-decide*:

```

assumes
  confl: conflicting  $S = \text{None}$  and
  inv: cdclW-M-level-inv  $S$  and
  n-s: no-step cdclW-merge-cp  $S$ 
shows no-step cdclW-s'-without-decide  $S$ 
proof (rule ccontr)
  assume  $\neg$  no-step cdclW-s'-without-decide  $S$ 
  then obtain  $T$  where
    cdclW: cdclW-s'-without-decide  $S$   $T$ 
  by auto
  then have inv-T: cdclW-M-level-inv  $T$ 
  using rtranclp-cdclW-s'-without-decide-rtranclp-cdclW[of  $S$   $T$ ]
  rtranclp-cdclW-consistent-inv inv by blast
  from cdclW show False
  proof cases
  case conflict'-without-decide
  have no-step propagate  $S$ 
  using n-s by blast
  then have conflict  $S$   $T$ 
  using local.conflict' tranclp-cdclW-cp-propagate-with-conflict-or-not[of  $S$   $T$ ]
  unfolding full1-def by (metis full1-def local.conflict'-without-decide rtranclp-unfold
    tranclp-unfold-begin)
  moreover
  then obtain  $T'$  where full cdclW-bj  $T$   $T'$ 
  using cdclW-bj-exists-normal-form inv-T by blast
  ultimately show False using cdclW-merge-cp.conflict' n-s by meson
next
case (bj'-without-decide  $S'$ )
then show ?thesis
  using confl unfolding full1-def by (fastforce simp: cdclW-bj.simps dest: tranclpD
    elim: rulesE)
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  $\neg$  ?thesis
  then obtain  $T$  where cdclW-merge-cp  $S$   $T$ 
  by auto
  then show False
  proof cases
  case (conflict'  $S'$ )
  then show False using n-s conflict'-without-decide conflict-is-full1-cdclW-cp by blast
next
case propagate'

```

moreover
 have $cdcl_W$ -all-struct-inv T
 using inv by ($meson$ $local.propagate'$ $rtranclp$ - $cdcl_W$ -all-struct-inv-inv
 $rtranclp$ -propagate-is- $rtranclp$ - $cdcl_W$ $tranclp$ -into- $rtranclp$)
 then obtain U where full $cdcl_W$ -cp T U
 using $cdcl_W$ -cp-normalized-element-all-inv by auto
 ultimately have full1 $cdcl_W$ -cp S U
 using $tranclp$ -full-full1I[of $cdcl_W$ -cp S T U] $cdcl_W$ -cp. $propagate'$
 $tranclp$ -mono[of $propagate$ $cdcl_W$ -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 $cdcl_W$ -merge-cp $S \implies cdcl_W$ -M-level-inv $S \implies no$ -step $cdcl_W$ -cp S
 using $cdcl_W$ -bj-exists-normal-form $cdcl_W$ -consistent-inv[OF $cdcl_W.conflict$, of S]
 by ($metis$ $cdcl_W$ -cp.cases $cdcl_W$ -merge-cp.simps $tranclp.intros(1)$)

lemma $conflicting$ -not-true- $rtranclp$ - $cdcl_W$ -merge-cp- no -step- $cdcl_W$ -bj:
 assumes
 $conflicting$ $S = None$ and
 $cdcl_W$ -merge-cp** S T
 shows no -step $cdcl_W$ -bj T
 using $assms(2,1)$ by (induction)
 (fastforce simp: $cdcl_W$ -merge-cp.simps full-def $tranclp$ -unfold-end $cdcl_W$ -bj.simps
 $elim$: $rulesE$) $+$

lemma $conflicting$ -true-full- $cdcl_W$ -merge-cp-iff-full- $cdcl_W$ -s'-without-decode:
 assumes
 $confl$: $conflicting$ $S = None$ and
 inv : $cdcl_W$ -all-struct-inv S
 shows
 full $cdcl_W$ -merge-cp S $V \iff full$ $cdcl_W$ -s'-without-decode S V (is $?fw \iff ?s'$)

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] $\langle ?fw \rangle$ unfolding full-def
 by (simp add: inv $rtranclp$ - $cdcl_W$ -all-struct-inv-inv)
 consider
 (s') $cdcl_W$ -s'-without-decode** S V
 | ($propa$) T where $cdcl_W$ -s'-without-decode** S T and $propagate^{++}$ T V
 | (bj) T U where $cdcl_W$ -s'-without-decode** 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-decode $confl$ st n -s by $metis$
 then have $cdcl_W$ -s'-without-decode** 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 trancpl-mono[of propagate cdclW-cp] cdclW-cp.propagate' unfolding full1-def
    by blast
  then have cdclW-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 cdclW-cp V
  using no-step-cdclW-merge-cp-no-step-cdclW-cp n-s inv-V
  unfolding cdclW-all-struct-inv-def by blast
then have full cdclW-cp U V
  using propa rtrancpl-mono[of propagate cdclW-cp] cdclW-cp.propagate' unfolding full-def
  by blast
moreover have no-step cdclW-cp T
  using bj unfolding full1-def by (fastforce dest!: trancplD simp:cdclW-bj.simps elim: rulesE)
ultimately have cdclW-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 cdclW-s'-without-decide V
proof (cases conflicting V = None)
case False
{ fix ss :: 'st
  have ff1:  $\forall s sa. \neg cdcl_W-s' s sa \vee full1\ cdcl_W-cp s sa$ 
     $\vee (\exists sb. decide\ s\ sb \wedge no\_step\ cdcl_W-cp\ s \wedge full\ cdcl_W-cp\ sb\ sa)$ 
     $\vee (\exists sb. full1\ cdcl_W-bj\ s\ sb \wedge no\_step\ cdcl_W-cp\ s \wedge full\ cdcl_W-cp\ sb\ sa)$ 
    by (metis cdclW-s'.cases)
  have ff2:  $(\forall p s sa. \neg full1\ p\ (s::'st)\ sa \vee p^{++}\ s\ sa \wedge no\_step\ p\ sa)$ 
     $\wedge (\forall p s sa. (\neg p^{++}\ (s::'st)\ sa \vee (\exists s. p\ sa\ s)) \vee full1\ p\ s\ sa)$ 
    by (meson full1-def)
  obtain ssa :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
    ff3:  $\forall p s sa. \neg p^{++}\ s\ sa \vee p\ s\ (ssa\ p\ s\ sa) \wedge p^{**}\ (ssa\ p\ s\ sa)\ sa$ 
    by (metis (no-types) trancplD)
  then have a3:  $\neg cdcl_W-cp^{++}\ V\ ss$ 
    using False by (metis option-full-cdclW-cp full-def)
  have  $\bigwedge s. \neg cdcl_W-bj^{++}\ V\ s$ 
    using ff3 False by (metis confl st
      conflicting-not-true-rtrancpl-cdclW-merge-cp-no-step-cdclW-bj)
  then have  $\neg cdcl_W-s'-without-decide\ V\ ss$ 
    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 simp
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 rtrancp-cdclW-s'-without-decide-rtrancp-cdclW st by blast
  then have inv-V: cdclW-all-struct-inv V using inv rtrancp-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 trancp-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 rtrancp-cdclW-all-struct-inv-inv trancp-into-rtrancp)
      then obtain X where full cdclW-cp W' X
        using cdclW-cp-normalized-element-all-inv by blast
      ultimately show False
        using bj'-without-decide n-s-cp-V n-s by blast
    qed
  from s' consider
    (cp-true) cdclW-merge-cp** S V and conflicting V = None
  | (cp-false) cdclW-merge-cp** S V and conflicting V ≠ None and no-step cdclW-cp V and
    no-step cdclW-bj V
  | (cp-conf) T where cdclW-merge-cp** S T conflict T V
  using rtrancp-cdclW-s'-without-decide-is-rtrancp-cdclW-merge-cp[of S V] confl
  unfolding full-def by meson
  then have cdclW-merge-cp** S V
  proof cases
    case cp-conf note S-T = this(1) and conf-V = this(2)
    have full cdclW-bj V V
      using conf-V n-s-bj unfolding full-def by fast
    then have cdclW-merge-cp T V
      using cdclW-merge-cp.conflict' conf-V by auto
    then show ?thesis using S-T by auto
  qed fast+
  moreover
    then have cdclW** S V using rtrancp-cdclW-merge-cp-rtrancp-cdclW by blast
    then have cdclW-all-struct-inv V
      using inv rtrancp-cdclW-all-struct-inv-inv by blast
    then have no-step cdclW-merge-cp V
      using conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp s'
      unfolding full-def by blast
    ultimately show ?fw unfolding full-def by auto
  qed

```

lemma *conflicting-true-full1-cdcl_W-merge-cp-iff-full1-cdcl_W-s'-without-decode:*
assumes

```

  confl: conflicting  $S = \text{None}$  and
  inv:  $\text{cdcl}_W\text{-all-struct-inv } S$ 
shows
   $\text{full1 } \text{cdcl}_W\text{-merge-cp } S \ V \longleftrightarrow \text{full1 } \text{cdcl}_W\text{-s'-without-decide } S \ V$ 
proof -
  have  $\text{full } \text{cdcl}_W\text{-merge-cp } S \ V = \text{full } \text{cdcl}_W\text{-s'-without-decide } S \ V$ 
    using confl conflicting-true-full-cdclW-merge-cp-iff-full-cdclW-s'-without-decode inv
    by simp
  then show ?thesis unfolding full-unfold full1-def
    by (metis (mono-tags) tranclp-unfold-begin)
qed

lemma conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode:
  assumes
    fw:  $\text{full1 } \text{cdcl}_W\text{-merge-cp } S \ V$  and
    inv:  $\text{cdcl}_W\text{-all-struct-inv } S$ 
  shows
     $\text{full1 } \text{cdcl}_W\text{-s'-without-decide } S \ V$ 
proof -
  have conflicting  $S = \text{None}$ 
    using fw unfolding full1-def by (auto dest!: tranclpD simp: cdclW-merge-cp.simps elim: rulesE)
  then show ?thesis
    using conflicting-true-full1-cdclW-merge-cp-iff-full1-cdclW-s'-without-decode fw inv by simp
qed

inductive cdclW-merge-stgy where
  fw-s-cp[intro]:  $\text{full1 } \text{cdcl}_W\text{-merge-cp } S \ T \implies \text{cdcl}_W\text{-merge-stgy } S \ T \mid$ 
  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 cdclW-merge-stgy-tranclp-cdclW-merge:
  assumes fw:  $\text{cdcl}_W\text{-merge-stgy } S \ T$ 
  shows  $\text{cdcl}_W\text{-merge}^{++} S \ T$ 
proof -
  { fix  $S \ T$ 
    assume  $\text{full1 } \text{cdcl}_W\text{-merge-cp } S \ T$ 
    then have  $\text{cdcl}_W\text{-merge}^{++} S \ T$ 
      using tranclp-mono[of  $\text{cdcl}_W\text{-merge-cp } \text{cdcl}_W\text{-merge}^{++}$ ]  $\text{cdcl}_W\text{-merge-cp-tranclp-cdcl}_W\text{-merge}$ 
      unfolding full1-def
      by auto
  } note full1-cdclW-merge-cp-cdclW-merge = this
  show ?thesis
    using fw
    apply (induction rule: cdclW-merge-stgy.induct)
    using full1-cdclW-merge-cp-cdclW-merge apply simp
    unfolding full-unfold by (auto dest!: full1-cdclW-merge-cp-cdclW-merge fw-decide)
qed

lemma rtranclp-cdclW-merge-stgy-rtranclp-cdclW-merge:
  assumes fw:  $\text{cdcl}_W\text{-merge-stgy}^{**} S \ T$ 
  shows  $\text{cdcl}_W\text{-merge}^{**} S \ T$ 
  using fw  $\text{cdcl}_W\text{-merge-stgy-tranclp-cdcl}_W\text{-merge}$  rtranclp-mono[of  $\text{cdcl}_W\text{-merge-stgy } \text{cdcl}_W\text{-merge}^{++}$ ]
  unfolding tranclp-rtranclp-rtranclp by blast

lemma cdclW-merge-stgy-rtranclp-cdclW:

```

$cdcl_W\text{-merge-stgy } S \ T \implies cdcl_W^{**} \ S \ T$
apply (induction rule: $cdcl_W\text{-merge-stgy.induct}$)
using $rtrancpl\text{-}cdcl_W\text{-merge-cp-rtrancpl-cdcl_W}$ **unfolding** $full1\text{-def}$
apply (simp add: $trancpl\text{-into-rtrancpl}$)
using $rtrancpl\text{-}cdcl_W\text{-merge-cp-rtrancpl-cdcl_W}$ $cdcl_W\text{-o.decide}$ $cdcl_W\text{-other}$ **unfolding** $full\text{-def}$
by (meson $r\text{-into-rtrancpl}$ $rtrancpl\text{-trans}$)

lemma $rtrancpl\text{-}cdcl_W\text{-merge-stgy-rtrancpl-cdcl_W}$:
 $cdcl_W\text{-merge-stgy}^{**} \ S \ T \implies cdcl_W^{**} \ S \ T$
using $rtrancpl\text{-mono}$ [of $cdcl_W\text{-merge-stgy}$ $cdcl_W^{**}$] $cdcl_W\text{-merge-stgy-rtrancpl-cdcl_W}$ **by** *auto*

lemma $cdcl_W\text{-merge-stgy-cases}$ [consumes 1, case-names $fw\text{-s-cp}$ $fw\text{-s-decide}$]:
assumes
 $cdcl_W\text{-merge-stgy } S \ U$
 $full1 \ cdcl_W\text{-merge-cp } S \ U \implies P$
 $\bigwedge T. \ decide \ S \ T \implies no\text{-step } cdcl_W\text{-merge-cp } S \implies full \ cdcl_W\text{-merge-cp } T \ U \implies P$
shows P
using *assms* **by** (auto simp: $cdcl_W\text{-merge-stgy.simps}$)

inductive $cdcl_W\text{-s'-w} :: 'st \Rightarrow 'st \Rightarrow bool$ **where**
 $conflict': full1 \ cdcl_W\text{-s'-without-decide } S \ S' \implies cdcl_W\text{-s'-w } S \ S' \mid$
 $decide': decide \ S \ S' \implies no\text{-step } cdcl_W\text{-s'-without-decide } S \implies full \ cdcl_W\text{-s'-without-decide } S' \ S''$
 $\implies cdcl_W\text{-s'-w } S \ S''$

lemma $cdcl_W\text{-s'-w-rtrancpl-cdcl_W}$:
 $cdcl_W\text{-s'-w } S \ T \implies cdcl_W^{**} \ S \ T$
apply (induction rule: $cdcl_W\text{-s'-w.induct}$)
using $rtrancpl\text{-}cdcl_W\text{-s'-without-decide-rtrancpl-cdcl_W}$ **unfolding** $full1\text{-def}$
apply (simp add: $trancpl\text{-into-rtrancpl}$)
using $rtrancpl\text{-}cdcl_W\text{-s'-without-decide-rtrancpl-cdcl_W}$ **unfolding** $full\text{-def}$
by (meson $decide$ $other$ $rtrancpl\text{-into-trancpl2}$ $trancpl\text{-into-rtrancpl}$)

lemma $rtrancpl\text{-}cdcl_W\text{-s'-w-rtrancpl-cdcl_W}$:
 $cdcl_W\text{-s'-w}^{**} \ S \ T \implies cdcl_W^{**} \ S \ T$
using $rtrancpl\text{-mono}$ [of $cdcl_W\text{-s'-w}$ $cdcl_W^{**}$] $cdcl_W\text{-s'-w-rtrancpl-cdcl_W}$ **by** *auto*

lemma $no\text{-step-cdcl_W-cp-no-step-cdcl_W-s'-without-decide}$:
assumes $no\text{-step } cdcl_W\text{-cp } S$ **and** $conflicting \ S = None$ **and** $inv: cdcl_W\text{-M-level-inv } S$
shows $no\text{-step } cdcl_W\text{-s'-without-decide } S$
by (metis *assms* $cdcl_W\text{-cp.conflict'}$ $cdcl_W\text{-cp.propagate'}$ $cdcl_W\text{-merge-restart-cases}$ $trancplD$ $conflicting\text{-true-no-step-cdcl_W-merge-cp-no-step-s'-without-decide}$)

lemma $no\text{-step-cdcl_W-cp-no-step-cdcl_W-merge-restart}$:
assumes $no\text{-step } cdcl_W\text{-cp } S$ **and** $conflicting \ S = None$
shows $no\text{-step } cdcl_W\text{-merge-cp } S$
by (metis *assms*(1) $cdcl_W\text{-cp.conflict'}$ $cdcl_W\text{-cp.propagate'}$ $cdcl_W\text{-merge-restart-cases}$ $trancplD$)

lemma $after\text{-}cdcl_W\text{-s'-without-decide-no-step-cdcl_W-cp}$:
assumes $cdcl_W\text{-s'-without-decide } S \ T$
shows $no\text{-step } cdcl_W\text{-cp } T$
using *assms* **by** (induction rule: $cdcl_W\text{-s'-without-decide.induct}$) (auto simp: $full1\text{-def}$ $full\text{-def}$)

lemma $no\text{-step-cdcl_W-s'-without-decide-no-step-cdcl_W-cp}$:
 $cdcl_W\text{-all-struct-inv } S \implies no\text{-step } cdcl_W\text{-s'-without-decide } S \implies no\text{-step } cdcl_W\text{-cp } S$
by (simp add: $conflicting\text{-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp}$ $no\text{-step-cdcl_W-merge-cp-no-step-cdcl_W-cp}$ $cdcl_W\text{-all-struct-inv-def}$)

lemma *after-cdcl_W-s'-w-no-step-cdcl_W-cp*:
assumes *cdcl_W-s'-w S T* **and** *cdcl_W-all-struct-inv S*
shows *no-step cdcl_W-cp T*
using *assms*
proof (*induction rule: cdcl_W-s'-w.induct*)
case *conflict'*
then show *?case*
by (*auto simp: full1-def tranclp-unfold-end after-cdcl_W-s'-without-decide-no-step-cdcl_W-cp*)
next
case (*decide' S T U*)
moreover
then have *cdcl_W** S U*
using *rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W[of T U] cdcl_W.other[of S T]*
cdcl_W-o.decide unfolding full-def by auto
then have *cdcl_W-all-struct-inv U*
using *decide'.prems rtranclp-cdcl_W-all-struct-inv-inv by blast*
ultimately show *?case*
using *no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-cp unfolding full-def by blast*
qed

lemma *rtranclp-cdcl_W-s'-w-no-step-cdcl_W-cp-or-eq*:
assumes *cdcl_W-s'-w** S T* **and** *cdcl_W-all-struct-inv S*
shows *S = T ∨ no-step cdcl_W-cp T*
using *assms*
proof (*induction rule: rtranclp-induct*)
case *base*
then show *?case by simp*
next
case (*step T U*)
moreover have *cdcl_W-all-struct-inv T*
using *rtranclp-cdcl_W-s'-w-rtranclp-cdcl_W[of S U] assms(2) rtranclp-cdcl_W-all-struct-inv-inv*
rtranclp-cdcl_W-s'-w-rtranclp-cdcl_W step.hyps(1) by blast
ultimately show *?case using after-cdcl_W-s'-w-no-step-cdcl_W-cp by fast*
qed

lemma *rtranclp-cdcl_W-merge-stgy'-no-step-cdcl_W-cp-or-eq*:
assumes *cdcl_W-merge-stgy** S T* **and** *inv: cdcl_W-all-struct-inv S*
shows *S = T ∨ no-step cdcl_W-cp T*
using *assms*
proof (*induction rule: rtranclp-induct*)
case *base*
then show *?case by simp*
next
case (*step T U*)
moreover have *cdcl_W-all-struct-inv T*
using *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W[of S U] assms(2) rtranclp-cdcl_W-all-struct-inv-inv*
rtranclp-cdcl_W-s'-w-rtranclp-cdcl_W step.hyps(1)
by (*meson rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W*)
ultimately show *?case*
using *after-cdcl_W-s'-w-no-step-cdcl_W-cp inv unfolding cdcl_W-all-struct-inv-def*
by (*metis cdcl_W-all-struct-inv-def cdcl_W-merge-stgy.simps full1-def full-def*
no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp rtranclp-cdcl_W-all-struct-inv-inv
rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W tranclp.intros(1) tranclp-into-rtranclp)
qed

lemma *no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj*:
assumes *no-step cdcl_W-s'-without-decide S and inv: cdcl_W-all-struct-inv S*
shows *no-step cdcl_W-bj S*
proof (*rule ccontr*)
assume $\neg ?thesis$
then obtain *T where S-T: cdcl_W-bj S T*
by *auto*
have *cdcl_W-all-struct-inv T*
using *S-T cdcl_W-all-struct-inv-inv inv other by blast*
then obtain *T' where full1 cdcl_W-bj S T'*
using *cdcl_W-bj-exists-normal-form[of T] full-full1 S-T unfolding cdcl_W-all-struct-inv-def*
by *metis*
moreover
then have *cdcl_W** S T'*
using *rtrancp-mono[of cdcl_W-bj cdcl_W] cdcl_W.other cdcl_W-o.bj trancp-into-rtrancp[of cdcl_W-bj]*
unfolding *full1-def by blast*
then have *cdcl_W-all-struct-inv T'*
using *inv rtrancp-cdcl_W-all-struct-inv-inv by blast*
then obtain *U where full cdcl_W-cp T' U*
using *cdcl_W-cp-normalized-element-all-inv by blast*
moreover have *no-step cdcl_W-cp S*
using *S-T by (auto simp: cdcl_W-bj.simps elim: rulesE)*
ultimately show *False*
using *assms cdcl_W-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 = None and
inv: cdcl_W-all-struct-inv R
shows (*cdcl_W-merge-stgy** R V \wedge conflicting V = None*)
 \vee (*cdcl_W-merge-stgy** R V \wedge conflicting V \neq None \wedge no-step cdcl_W-bj V*)
 \vee ($\exists S T U. \text{cdcl}_W\text{-merge-stgy** } R S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S T$
 $\wedge \text{cdcl}_W\text{-merge-cp** } T U \wedge \text{conflict } U V$)


```

  ∨ (∃ S T. cdclW-merge-stgy** R S ∧ no-step cdclW-merge-cp S ∧ decide S T
    ∧ cdclW-merge-cp** T V
    ∧ conflicting V = None)
  ∨ (cdclW-merge-cp** R V ∧ conflicting V = None)
  ∨ (∃ U. cdclW-merge-cp** R U ∧ 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') 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 = None
    | (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
  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 = None
    | (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 meson
  then show ?thesis
  proof cases
    case V-W
    then show ?thesis using ⟨R = V⟩ n-s-R by simp
  next
    case propa
    then show ?thesis using ⟨R = V⟩ 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' ⟨R = V⟩ by blast
  qed
next
  case dec-conf note - = this(5)
  then have False using conflict' unfolding full1-def by (auto dest!: tranclpD elim: rulesE)

```

```

    then show ?thesis by fast
next
case dec note T-V = this(4)
consider
  (propa) propagate++ V W and conflicting W = None
  | (propa-confl) 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 meson
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-confl
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-confl(2) by metis
qed
next
case cp
consider
  (propa) propagate++ V W and conflicting W = None
  | (propa-confl) 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 meson
then show ?thesis
proof cases
  case propa
  then show ?thesis by (meson cdclW-merge-cp.propagate' cp
    rtranclp.rtrancl-into-rtrancl)
next
case propa-confl
then show ?thesis
  using propa-confl(2) cp
  by (metis (full-types) cdclW-merge-cp.propagate' rtranclp.rtrancl-into-rtrancl
    rtranclp-unfold)
qed
next
case cp-confl
then show ?thesis using conflict' unfolding full1-def by (fastforce dest!: tranclpD
  elim!: rulesE)
qed
next
case (decide' V')
then have conf-V: conflicting V = None
  by (auto elim: rulesE)
consider
  (s') cdclW-merge-stgy** R V
  | (dec-confl) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (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 = None
  | (cp) cdclW-merge-cp** R V
  | (cp-confl) U where cdclW-merge-cp** R U and conflict U V

```

```

using IH by meson
then show ?thesis
proof cases
case s'
have confl-V': conflicting V' = None using decide'(1) by (auto elim: rulesE)
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 = None
| (propa-confl) 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 confl-V' local.decide'(1,2) s' conf-V
no-step-cdclW-cp-no-step-cdclW-merge-restart[of V] by (auto elim: rulesE)
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-confl
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-confl(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 = None
| (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 meson
then show ?thesis
proof cases
case V'-W
moreover have conflicting V' = None

```

```

    using decide'(1) by (auto elim: rulesE)
  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-rtrancpl)
next
case propa-confl
moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtrancpl-unfold trancpl-unfold-end)
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtrancpl)
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 auto
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 = None
| (propa-confl) V'' where propagate** V' V'' and conflict V'' W
using trancpl-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
unfolding full-unfold full1-def by meson
then show ?thesis

proof cases
case V'-W
moreover have conflicting V' = None
  using decide'(1) by (auto elim: rulesE)
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-rtrancpl)
next
case propa-confl
moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtrancpl-unfold trancpl-unfold-end)
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtrancpl)
qed

```

```

next
  case (dec-conf)
  show ?thesis using conf-V dec-conf(5) by (auto elim!: rulesE
    simp del: state-simp simp: state-eq-def)
next
  case cp-conf
  then show ?thesis using decide' apply - by (intro HOL.disjI2) (fastforce elim: rulesE
    simp del: state-simp simp: state-eq-def)
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 = None
| (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 = None
| (cp) cdclW-merge-cp** R V and conflicting V = None
| (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
      elim: rulesE)
  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
      elim: rulesE)
  then show ?thesis by fast
next
  case dec-conf
  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-conf(4) by simp
  consider
    (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = None
  | (propa-conf) 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 meson
  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 = None
  then show ?thesis using ⟨cdclW-merge-stgy** R V'⟩ ⟨V' = W⟩ by auto
next
  assume conflicting W ≠ None
  then show ?thesis
    using ⟨cdclW-merge-stgy** R V'⟩ ⟨V' = W⟩ by (metis ⟨cdclW-merge-cp U V'⟩
      conflictE conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj
      dec-confl(5) map-option-is-None r-into-rtranclp)
  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-confl
  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 = None
  | (propa-confl) 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 meson
  then show ?thesis

proof cases
  case V'-W
  show ?thesis
  proof cases
    assume conflicting V' = None
    then show ?thesis
      using V'-W ⟨cdclW-merge-cp U V'⟩ cp-confl(1) by force
  next

```

```

    assume confl: conflicting  $V' \neq \text{None}$ 
    then have no-step cdclW-merge-stgy  $V'$ 
      by (fastforce simp: cdclW-merge-stgy.simps full1-def full-def
          cdclW-merge-cp.simps dest!: tranclpD elim: rulesE)
    have no-step cdclW-merge-cp  $V'$ 
      using confl by (auto simp: full1-def full-def cdclW-merge-cp.simps
          dest!: tranclpD elim: rulesE)
    moreover have cdclW-merge-cp  $U\ W$ 
      using  $V'-W \langle \text{cdcl}_W\text{-merge-cp } U\ V' \rangle$  by blast
    ultimately have full1 cdclW-merge-cp  $R\ V'$ 
      using cp-confl(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  $\langle \text{no-step cdcl}_W\text{-merge-cp } V' \rangle$  by (auto simp: cdclW-merge-stgy.simps
          full1-def dest!: tranclpD elim: rulesE)
    ultimately have cdclW-merge-stgy**  $R\ V'$  by auto
    show ?thesis by (metis  $V'-W \langle \text{cdcl}_W\text{-merge-cp } U\ V' \rangle \langle \text{cdcl}_W\text{-merge-stgy** } R\ V' \rangle$ 
        conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj cp-confl(1)
        rtranclp.rtrancl-into-rtrancl step.premis)
  qed
next
  case propa
  moreover then have cdclW-merge-cp  $V'\ W$ 
    by auto
  ultimately show ?thesis using  $\langle \text{cdcl}_W\text{-merge-cp } U\ V' \rangle$  cp-confl(1) by force
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  $\langle \text{cdcl}_W\text{-merge-cp } U\ V' \rangle$  cp-confl(1) by (metis rtranclp.rtrancl-into-rtrancl
        rtranclp-trans)
  qed
qed
qed
qed

```

lemma *decide-rtranclp-cdcl_W-s'-rtranclp-cdcl_W-s'*:

assumes

dec: *decide* $S\ T$ **and**

*cdcl_W-s'^{***}* $T\ U$ **and**

n-s-S: *no-step cdcl_W-cp* S **and**

no-step cdcl_W-cp U

shows *cdcl_W-s'^{***}* $S\ U$

using *assms(2,4)*

proof *induction*

case (*step* $U\ V$) **note** $st = \text{this}(1)$ **and** $s' = \text{this}(2)$ **and** $IH = \text{this}(3)$ **and** $n-s = \text{this}(4)$

consider

$(TU)\ T = U$

| $(s'-st)\ T'$ **where** *cdcl_W-s'* $T\ T'$ **and** *cdcl_W-s'^{***}* $T'\ U$

using $st[\text{unfolded } rtranclp\text{-unfold}]$ **by** (auto *dest!: tranclpD*)

then show ?*case*

proof *cases*

case TU

```

then show ?thesis
proof -
  assume a1:  $T = U$ 
  then have f2:  $cdcl_W-s' T V$ 
    using  $s'$  by force
  obtain ss :: ' $st$  where
    ss:  $cdcl_W-s'^{**} S T \vee cdcl_W-cp T ss$ 
    using a1 step.IH by blast-
  obtain ssa :: ' $st \Rightarrow 'st$  where
    f3:  $\forall s sa sb. (\neg decide s sa \vee cdcl_W-cp s (ssa s) \vee \neg full cdcl_W-cp sa sb) \vee cdcl_W-s' s sb$ 
    using  $cdcl_W-s'.decide'$  by moura
  have  $\forall s sa. \neg cdcl_W-s' s sa \vee full1 cdcl_W-cp s sa \vee (\exists sb. decide s sb \wedge no-step cdcl_W-cp s \wedge full cdcl_W-cp sb sa) \vee (\exists sb. full1 cdcl_W-bj s sb \wedge no-step cdcl_W-cp s \wedge full cdcl_W-cp sb sa)$ 
    by (metis  $cdcl_W-s'E$ )
  then have  $\exists s. cdcl_W-s'^{**} S s \wedge cdcl_W-s' s V$ 
    using f3 ss f2 by (metis dec full1-is-full n-s-S rtranclp-unfold)
  then show ?thesis
    by force
qed
next
case ( $s'-st T'$ ) note  $s'-T' = this(1)$  and  $st = this(2)$ 
have  $cdcl_W-s'^{**} S T'$ 
  using  $s'-T'$ 
proof cases
  case conflict'
  then have  $cdcl_W-s' S T'$ 
    using dec  $cdcl_W-s'.decide' n-s-S$  by (simp add: full-unfold)
  then show ?thesis
    using st by auto
next
  case ( $decide' T''$ )
  then have  $cdcl_W-s' S T$ 
    using dec  $cdcl_W-s'.decide' n-s-S$  by (simp add: full-unfold)
  then show ?thesis using  $decide' s'-T'$  by auto
next
  case bj'
  then have False
    using dec unfolding full1-def by (fastforce dest!: tranclpD simp:  $cdcl_W-bj.simps$  elim: rulesE)
  then show ?thesis by fast
qed
then show ?thesis using  $s' st$  by auto
qed
next
case base
then have  $full cdcl_W-cp T T$ 
  by (simp add: full-unfold)
then show ?case
  using  $cdcl_W-s'.simps$  dec n-s-S by auto
qed

lemma rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-s':
  assumes

```



```

    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 rtrancp-cdclW-all-struct-inv-inv rtrancp-cdclW-merge-stgy-rtrancp-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 ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st where
        f2: ∧ p s sa pa sb sc sd pb se sf. (¬ full1 p (s::'st) sa ∨ p++ s sa)
          ∧ (¬ pa (sb::'st) sc ∨ ¬ full1 pa sd sb) ∧ (¬ pb++ se sf ∨ pb sf (ss pb sf)
          ∨ 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 ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
        f4: ∧ p s sa. ¬ p++ s sa ∨ p s (ssa p s sa)
      by (meson trancp-unfold-begin)
      then have f5: ∧ s. ¬ full1 cdclW-merge-cp s S
      using f3 f2 by (metis (full-types))
      have ∧ s. ¬ full cdclW-merge-cp s S
      using f4 f3 by (meson full-def)
      then have S = R
      using f5 by (metis cdclW-cp.conflict' cdclW-cp.propagate' cdclW-merge-cp.cases f3 f4 inv
        rtrancp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq st)
      then show ?thesis
      using f2 a1 by (metis (no-types) ⟨cdclW-all-struct-inv S⟩
        conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode
        rtrancp-cdclW-s'-without-decide-rtrancp-cdclW-s' rtrancp-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' = None
      by (auto elim: rulesE)
    ultimately have full cdclW-s'-without-decide S' T
      by (meson ⟨cdclW-all-struct-inv S⟩ cdclW-merge-restart-cdclW fw-r-decide
        rtrancp-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) rtrancp-cdclW-s'-without-decide-rtrancp-cdclW-s')
    have cdclW-merge-stgy** S T
      using fw by blast
    then have cdclW-s'** S T
      using decide-rtrancp-cdclW-s'-rtrancp-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
        rtrancp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
  qed

```

then show ?thesis using IH by auto
qed
qed

lemma *rtranclp-cdcl_W-merge-stgy-distinct-mset-clauses:*

assumes *invR: cdcl_W-all-struct-inv R and*
*st: cdcl_W-merge-stgy** R S and*
dist: distinct-mset (clauses R) and
R: trail R = []
shows *distinct-mset (clauses S)*
using *rtranclp-cdcl_W-stgy-distinct-mset-clauses[OF invR - dist R]*
*invR st rtranclp-mono[of cdcl_W-s' cdcl_W-stgy**] cdcl_W-s'-is-rtranclp-cdcl_W-stgy*
by *(auto dest!: cdcl_W-s'-is-rtranclp-cdcl_W-stgy rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-s')*

lemma *no-step-cdcl_W-s'-no-step-cdcl_W-merge-stgy:*

assumes
inv: cdcl_W-all-struct-inv R and s': no-step cdcl_W-s' R
shows *no-step cdcl_W-merge-stgy R*

proof –

{ **fix** *ss :: 'st*
obtain *ssa :: 'st ⇒ 'st ⇒ 'st where*
ff1: ∧ s sa. ¬ cdcl_W-merge-stgy s sa ∨ full1 cdcl_W-merge-cp s sa ∨ decide s (ssa s sa)
using *cdcl_W-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 tranclp-unfold-begin)*
obtain *ssc :: 'st ⇒ 'st where*
ff3: ∧ s sa sb. (¬ cdcl_W-all-struct-inv s ∨ ¬ cdcl_W-cp s sa ∨ cdcl_W-s' s (ssc s))
∧ (¬ cdcl_W-all-struct-inv s ∨ ¬ cdcl_W-o s sb ∨ cdcl_W-s' s (ssc s))
using *n-step-cdcl_W-stgy-iff-no-step-cdcl_W-cl-cdcl_W-o* **by** *moura*
then have *ff4: ∧ s. ¬ cdcl_W-o R s*
using *s' inv* **by** *blast*
have *ff5: ∧ s. ¬ cdcl_W-cp⁺⁺ R s*
using *ff3 ff2 s'* **by** *(metis inv)*
have *∧ s. ¬ cdcl_W-bj⁺⁺ R s*
using *ff4 ff2* **by** *(metis bj)*
then have *∧ s. ¬ cdcl_W-s'-without-decide R s*
using *ff5* **by** *(simp add: cdcl_W-s'-without-decide.simps full1-def)*
then have *¬ cdcl_W-s'-without-decide⁺⁺ R ss*
using *ff2* **by** *blast*
then have *¬ cdcl_W-merge-stgy R ss*
using *ff4 ff1* **by** *(metis (full-types) decide full1-def inv*
conflicting-true-full1-cdcl_W-merge-cp-imp-full1-cdcl_W-s'-without-decode) }
then show *?thesis*
by *fastforce*

qed

end

We will discharge the assumption later

locale *conflict-driven-clause-learning_W-termination =*

conflict-driven-clause-learning_W +
assumes *wf-cdcl_W-merge: wf {(T, S). cdcl_W-all-struct-inv S ∧ cdcl_W-merge S T}*
begin

lemma *wf-tranclp-cdcl_W-merge: wf {(T, S). cdcl_W-all-struct-inv S ∧ cdcl_W-merge⁺⁺ S T}*

```

using wf-trancl[OF wf-cdclW-merge]
apply (rule wf-subset)
by (auto simp: trancl-set-tranclp
  cdclW-all-struct-inv-tranclp-cdclW-merge-tranclp-cdclW-merge-cdclW-all-struct-inv)

lemma wf-cdclW-merge-cp:
  wf{(T, S). cdclW-all-struct-inv S ∧ cdclW-merge-cp S T}
  using wf-tranclp-cdclW-merge by (rule wf-subset) (auto simp: cdclW-merge-cp-tranclp-cdclW-merge)

lemma wf-cdclW-merge-stgy:
  wf{(T, S). cdclW-all-struct-inv S ∧ cdclW-merge-stgy S T}
  using wf-tranclp-cdclW-merge by (rule wf-subset)
  (auto simp add: cdclW-merge-stgy-tranclp-cdclW-merge)

lemma cdclW-merge-cp-obtain-normal-form:
  assumes inv: cdclW-all-struct-inv R
  obtains S where full cdclW-merge-cp R S
proof –
  obtain S where full (λS T. cdclW-all-struct-inv S ∧ cdclW-merge-cp S T) R S
  using wf-exists-normal-form-full[OF wf-cdclW-merge-cp] by blast
  then have
    st: (λS T. cdclW-all-struct-inv S ∧ cdclW-merge-cp S T)** R S and
    n-s: no-step (λS T. cdclW-all-struct-inv S ∧ cdclW-merge-cp S T) S
  unfolding full-def by blast+
  have cdclW-merge-cp** R S
  using st by induction auto
  moreover
  have cdclW-all-struct-inv S
  using st inv
  apply (induction rule: rtranclp-induct)
  apply simp
  by (meson r-into-rtranclp rtranclp-cdclW-all-struct-inv-inv
    rtranclp-cdclW-merge-cp-rtranclp-cdclW)
  then have no-step cdclW-merge-cp S
  using n-s by auto
  ultimately show ?thesis
  using that unfolding full-def by blast
qed

lemma no-step-cdclW-merge-stgy-no-step-cdclW-s':
  assumes
    inv: cdclW-all-struct-inv R and
    confl: conflicting R = None and
    n-s: no-step cdclW-merge-stgy R
  shows no-step cdclW-s' R
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain S where cdclW-s' R S by auto
  then show False
  proof cases
  case conflict'
  then obtain S' where full1 cdclW-merge-cp R S'
  by (metis (full-types) cdclW-merge-cp-obtain-normal-form cdclW-s'-without-decide.simps confl
    conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide full-def full-unfold inv
    cdclW-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 auto
qed
qed

lemma rtranclp-cdclW-merge-cp-no-step-cdclW-bj:
  assumes conflicting R = None 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 auto

lemma rtranclp-cdclW-merge-stgy-no-step-cdclW-bj:
  assumes confl: conflicting R = None 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 elim: rulesE)
next
case (step S T) note st = this(1) and fw = this(2) and IH = this(3)
have confl-S: conflicting S = None
  using fw apply cases
  by (auto simp: full1-def cdclW-merge-cp.simps dest!: tranclpD elim: rulesE)
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' = None by (auto elim: rulesE)
ultimately show ?thesis
  using conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj
  unfolding full-def by meson
qed
qed

end

end

```

```

theory CDCL-W-Restart
imports CDCL-W-Merge
begin

```

20.5 Adding Restarts

```

locale cdclW-restart =
  conflict-driven-clause-learningW
  — functions for clauses:
  mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss

  — functions for the conflicting clause:
  mset-ccls union-ccls insert-ccls remove-clit

  — conversion
  ccls-of-cls cls-of-ccls

  — functions for the state:
  — access functions:
  trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
  — changing state:
  cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — get state:
  init-state
  restart-state
for
  mset-cls:: 'cls ⇒ 'v clause and
  union-cls :: 'cls ⇒ 'cls ⇒ 'cls and
  insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls and
  remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls and

  mset-clss:: 'clss ⇒ 'v clauses and
  union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
  in-clss :: 'cls ⇒ 'clss ⇒ bool and
  insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and

  mset-ccls:: 'ccls ⇒ 'v clause and
  union-ccls :: 'ccls ⇒ 'ccls ⇒ 'ccls and
  insert-ccls :: 'v literal ⇒ 'ccls ⇒ 'ccls and
  remove-clit :: 'v literal ⇒ 'ccls ⇒ 'ccls and

  ccls-of-cls :: 'cls ⇒ 'ccls and
  cls-of-ccls :: 'ccls ⇒ 'cls and

  trail :: 'st ⇒ ('v, nat, 'v clause) marked-lits and
  hd-raw-trail :: 'st ⇒ ('v, nat, 'cls) marked-lit and
  raw-init-clss :: 'st ⇒ 'clss and
  raw-learned-clss :: 'st ⇒ 'clss and
  backtrack-lvl :: 'st ⇒ nat and
  raw-conflicting :: 'st ⇒ 'ccls option and

  cons-trail :: ('v, nat, 'cls) marked-lit ⇒ 'st ⇒ 'st and

```

```

tl-trail :: 'st ⇒ 'st and
add-init-cls :: 'cls ⇒ 'st ⇒ 'st and
add-learned-cls :: 'cls ⇒ 'st ⇒ 'st and
remove-cls :: 'cls ⇒ 'st ⇒ 'st and
update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
update-conflicting :: 'ccls option ⇒ 'st ⇒ 'st and

init-state :: 'clss ⇒ 'st and
restart-state :: 'st ⇒ 'st +
fixes f :: nat ⇒ nat
assumes f: unbounded f
begin

```

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive *cdcl_W-merge-with-restart* **where**

restart-step:

```

(cdeclW-merge-stgy  $\sim$  (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 ⇒ cdeclW-merge-with-restart (S, n) (U, Suc n) |

```

restart-full: full1 cdecl_W-merge-stgy S T ⇒ cdecl_W-merge-with-restart (S, n) (T, Suc n)

lemma cdecl_W-merge-with-restart S T ⇒ cdecl_W-merge-restart** (fst S) (fst T)

by (induction rule: cdecl_W-merge-with-restart.induct)

```

(auto dest!: relpowp-imp-rtranclp cdeclW-merge-stgy-tranclp-cdeclW-merge tranclp-into-rtranclp
  rtranclp-cdeclW-merge-stgy-rtranclp-cdeclW-merge rtranclp-cdeclW-merge-tranclp-cdeclW-merge-restart
  fw-r-rf cdeclW-rf.restart
simp: full1-def)

```

lemma cdecl_W-merge-with-restart-rtranclp-cdecl_W:

cdecl_W-merge-with-restart S T ⇒ cdecl_W** (fst S) (fst T)

by (induction rule: cdecl_W-merge-with-restart.induct)

```

(auto dest!: relpowp-imp-rtranclp rtranclp-cdeclW-merge-stgy-rtranclp-cdeclW cdeclW.rf
  cdeclW-rf.restart tranclp-into-rtranclp simp: full1-def)

```

lemma cdecl_W-merge-with-restart-increasing-number:

cdecl_W-merge-with-restart S T ⇒ snd T = 1 + snd S

by (induction rule: cdecl_W-merge-with-restart.induct) auto

lemma full1 cdecl_W-merge-stgy S T ⇒ cdecl_W-merge-with-restart (S, n) (T, Suc n)

using restart-full **by** blast

lemma cdecl_W-all-struct-inv-learned-clss-bound:

assumes inv: cdecl_W-all-struct-inv S

shows set-mset (learned-clss S) ⊆ simple-clss (atms-of-mm (init-clss S))

proof

fix C

assume C: C ∈ set-mset (learned-clss S)

have distinct-mset C

using C inv **unfolding** cdecl_W-all-struct-inv-def distinct-cdecl_W-state-def distinct-mset-set-def

by auto

moreover **have** ¬tautology C

using C inv **unfolding** cdecl_W-all-struct-inv-def cdecl_W-learned-clause-def **by** auto

moreover

```

have atms-of  $C \subseteq \text{atms-of-mm } (\text{learned-clss } S)$ 
  using  $C$  by auto
then have atms-of  $C \subseteq \text{atms-of-mm } (\text{init-clss } S)$ 
  using inv unfolding cdclW-all-struct-inv-def no-strange-atm-def by force
moreover have finite (atms-of-mm (init-clss  $S$ ))
  using inv unfolding cdclW-all-struct-inv-def by auto
ultimately show  $C \in \text{simple-clss } (\text{atms-of-mm } (\text{init-clss } S))$ 
  using distinct-mset-not-tautology-implies-in-simple-clss simple-clss-mono
  by blast
qed

```

lemma *cdcl_W-merge-with-restart-init-clss*:

```

cdclW-merge-with-restart  $S \ T \implies \text{cdcl}_W\text{-M-level-inv } (\text{fst } S) \implies$ 
init-clss (fst  $S$ ) = init-clss (fst  $T$ )
using cdclW-merge-with-restart-rtrancpl-cdclW rtrancpl-cdclW-init-clss by blast

```

lemma

```

wf {( $T, S$ ). cdclW-all-struct-inv (fst  $S$ )  $\wedge$  cdclW-merge-with-restart  $S \ T$ }
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $g$  where
     $g: \bigwedge i. \text{cdcl}_W\text{-merge-with-restart } (g \ i) \ (g \ (\text{Suc } i))$  and
     $\text{inv}: \bigwedge i. \text{cdcl}_W\text{-all-struct-inv } (\text{fst } (g \ i))$ 
  unfolding wf-iff-no-infinite-down-chain by fast
  { fix  $i$ 
    have init-clss (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-mm (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 le-iff-add)

  obtain  $k$  where
     $f\text{-}g\text{-}k: f \ (\text{snd } (g \ k)) > \text{card } (\text{simple-clss } (\text{atms-of-mm } (\text{init-clss } (\text{fst } ?S))))$  and
     $k > \text{card } (\text{simple-clss } (\text{atms-of-mm } (\text{init-clss } (\text{fst } ?S))))$ 
    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  $g[\text{of } i]$ 
  have False
    proof (induction rule: cdclW-merge-with-restart.induct)
      case (restart-step  $T \ S \ n$ ) note  $H = \text{this}(1)$  and  $c = \text{this}(2)$  and  $n\text{-}s = \text{this}(4)$ 

```

```

    obtain  $S'$  where  $cdcl_W$ -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  $T (\text{fst } (g (k+1)))$  and
   $cdcl_W$ -merge-stgy: ( $cdcl_W$ -merge-stgy  $\sim m$ ) ( $\text{fst } (g k)$ )  $T$ 
  using  $g[\text{of } k] H[\text{of } \text{Suc } k]$  by (force simp:  $cdcl_W$ -merge-with-restart.simps full1-def)
have  $cdcl_W$ -merge-stgy** ( $\text{fst } (g k)$ )  $T$ 
  using  $cdcl_W$ -merge-stgy relpowp-imp-rtranclp by metis
then have  $cdcl_W$ -all-struct-inv  $T$ 
  using inv[ $\text{of } k$ ] rtranclp- $cdcl_W$ -all-struct-inv-inv rtranclp- $cdcl_W$ -merge-stgy-rtranclp- $cdcl_W$ 
  by blast
moreover have  $\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss} (\text{fst } (g k))))$ 
   $> \text{card} (\text{simple-clss} (\text{atms-of-mm} (\text{init-clss} (\text{fst } ?S))))$ 
  unfolding  $m[\text{symmetric}]$  using  $\langle m > f (\text{snd } (g k)) \rangle f-g-k$  by linarith
then have  $\text{card} (\text{set-mset} (\text{learned-clss } T))$ 
   $> \text{card} (\text{simple-clss} (\text{atms-of-mm} (\text{init-clss} (\text{fst } ?S))))$ 
  by linarith
moreover
  have  $\text{init-clss} (\text{fst } (g k)) = \text{init-clss } T$ 
    using  $\langle cdcl_W$ -merge-stgy** ( $\text{fst } (g k)$ )  $T \rangle$  rtranclp- $cdcl_W$ -merge-stgy-rtranclp- $cdcl_W$ 
    rtranclp- $cdcl_W$ -init-clss inv unfolding  $cdcl_W$ -all-struct-inv-def by blast
  then have  $\text{init-clss} (\text{fst } ?S) = \text{init-clss } T$ 
    using  $\text{init-g}[\text{of } k]$  by auto
ultimately show False
  using  $cdcl_W$ -all-struct-inv-learned-clss-bound
  by (simp add:  $\langle \text{finite} (\text{atms-of-mm} (\text{init-clss} (\text{fst } (g 0)))) \rangle$  simple-clss-finite
    card-mono leD)
qed

lemma  $cdcl_W$ -merge-with-restart-distinct-mset-clauses:
  assumes invR:  $cdcl_W$ -all-struct-inv ( $\text{fst } R$ ) and
  st:  $cdcl_W$ -merge-with-restart  $R S$  and
  dist: distinct-mset (clauses ( $\text{fst } R$ )) and
  R: trail ( $\text{fst } R$ ) = []
  shows distinct-mset (clauses ( $\text{fst } S$ ))
  using assms(2,1,3,4)
proof (induction)
  case (restart-full  $S T$ )
  then show ?case using rtranclp- $cdcl_W$ -merge-stgy-distinct-mset-clauses[ $\text{of } S T$ ] unfolding full1-def
    by (auto dest: tranclp-into-rtranclp)
next
  case (restart-step  $T S n U$ )
  then have distinct-mset (clauses  $T$ )
    using rtranclp- $cdcl_W$ -merge-stgy-distinct-mset-clauses[ $\text{of } S T$ ] unfolding full1-def
    by (auto dest: relpowp-imp-rtranclp)
  then show ?case using  $\langle \text{restart } T U \rangle$  by (metis clauses-restart distinct-mset-union fstI
    mset-le-exists-conv restart.cases state-eq-clauses)

```


qed

inductive *cdcl_W-with-restart* **where**

restart-step:

$(\text{cdcl}_W\text{-stgy} \sim (\text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } S)))) S T \implies$
 $\text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } S)) > f n \implies$
 $\text{restart } T U \implies$

$\text{cdcl}_W\text{-with-restart } (S, n) (U, \text{Suc } n) \mid$

restart-full: $\text{full1 } \text{cdcl}_W\text{-stgy } S T \implies \text{cdcl}_W\text{-with-restart } (S, n) (T, \text{Suc } n)$

lemma *cdcl_W-with-restart-rtranclp-cdcl_W*:

$\text{cdcl}_W\text{-with-restart } S T \implies \text{cdcl}_W^{**} (\text{fst } S) (\text{fst } T)$

apply (*induction rule*: *cdcl_W-with-restart.induct*)

by (*auto dest!*: *relopw-imp-rtranclp tranclp-into-rtranclp fw-r-rf*

cdcl_W-rf.restart rtranclp-cdcl_W-stgy-rtranclp-cdcl_W cdcl_W-merge-restart-cdcl_W

simp: *full1-def*)

lemma *cdcl_W-with-restart-increasing-number*:

$\text{cdcl}_W\text{-with-restart } S T \implies \text{snd } T = 1 + \text{snd } S$

by (*induction rule*: *cdcl_W-with-restart.induct*) *auto*

lemma *full1 cdcl_W-stgy* $S T \implies \text{cdcl}_W\text{-with-restart } (S, n) (T, \text{Suc } n)$

using *restart-full* **by** *blast*

lemma *cdcl_W-with-restart-init-clss*:

$\text{cdcl}_W\text{-with-restart } S T \implies \text{cdcl}_W\text{-M-level-inv } (\text{fst } S) \implies \text{init-clss } (\text{fst } S) = \text{init-clss } (\text{fst } T)$

using *cdcl_W-with-restart-rtranclp-cdcl_W rtranclp-cdcl_W-init-clss* **by** *blast*

lemma

$\text{wf } \{(T, S). \text{cdcl}_W\text{-all-struct-inv } (\text{fst } S) \wedge \text{cdcl}_W\text{-with-restart } S T\}$

proof (*rule ccontr*)

assume $\neg ?thesis$

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 $\text{init-clss } (\text{fst } (g \ i)) = \text{init-clss } (\text{fst } (g \ 0))$

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-mm* ($\text{init-clss } (\text{fst } ?S)$))

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*: $\text{unbounded } (\lambda i. f \ (\text{snd } (g \ i)))$

using *f unfolding bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g not-bounded-nat-exists-larger not-le le-iff-add*)

obtain k where

$f\text{-}g\text{-}k$: $f \text{ (snd (g k)) } > \text{card (simple-clss (atms-of-mm (init-clss (fst ?S))))}$ and
 $k > \text{card (simple-clss (atms-of-mm (init-clss (fst ?S))))}$
 using *not-bounded-nat-exists-larger*[*OF unbounded-f-g*] by *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

```

{ fix i
  assume no-step cdclW-stgy (fst (g i))
  with g[of i]
  have False
  proof (induction rule: cdclW-with-restart.induct)
    case (restart-step T S n) note H = this(1) and c = this(2) and n-s = this(4)
    obtain S' where cdclW-stgy S S'
    using H c by (metis gr-implies-not0 relpowp-E2)
    then show False using n-s by auto
  next
    case (restart-full S T)
    then show False unfolding full1-def by (auto dest: tranclpD)
  qed
} note H = this
obtain m T where
  m: m = card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k)))) and
  m > f (snd (g k)) and
  restart T (fst (g (k+1))) and
  cdclW-merge-stgy: (cdclW-stgy  $\sim$  m) (fst (g k)) T
  using g[of k] H[of Suc k] by (force simp: cdclW-with-restart.simps full1-def)
have cdclW-stgy** (fst (g k)) T
  using cdclW-merge-stgy relpowp-imp-rtranclp by metis
then have cdclW-all-struct-inv T
  using inv[of k] rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW by blast
moreover have card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k))))
  > card (simple-clss (atms-of-mm (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 (simple-clss (atms-of-mm (init-clss (fst ?S))))
  by linarith
moreover
  have init-clss (fst (g k)) = init-clss T
    using cdclW-stgy** (fst (g k)) T rtranclp-cdclW-stgy-rtranclp-cdclW rtranclp-cdclW-init-clss
    inv unfolding cdclW-all-struct-inv-def
    by blast
  then have init-clss (fst ?S) = init-clss T
    using init-g[of k] by auto
ultimately show False
  using cdclW-all-struct-inv-learned-clss-bound
  by (simp add: finite (atms-of-mm (init-clss (fst (g 0)))) simple-clss-finite
    card-mono leD)
qed

```

lemma *cdcl_W-with-restart-distinct-mset-clauses*:

assumes *invR*: *cdcl_W-all-struct-inv* (fst *R*) and
st: *cdcl_W-with-restart* *R S* and
dist: *distinct-mset* (*clauses* (fst *R*)) and
R: *trail* (fst *R*) = []
shows *distinct-mset* (*clauses* (fst *S*))

```

using assms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtrancpl-cdclW-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: trancpl-into-rtrancpl)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T) using rtrancpl-cdclW-stgy-distinct-mset-clauses[of S T]
    unfolding full1-def by (auto dest: relpowp-imp-rtrancpl)
  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  $\exists k :: \text{nat}. (2^{k-1} \leq i \wedge i < 2^k - 1) \vee i = 2^k - 1$ 
proof (induction i)
  case 0
  then show ?case
    by (rule exI[of - 0], simp)
next
  case (Suc n)
  then obtain k where  $2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1$ 
    by blast
  then consider
    (st-interv)  $2^{k-1} \leq n$  and  $n \leq 2^k - 2$ 
  | (end-interv)  $2^{k-1} \leq n$  and  $n = 2^k - 2$ 
  | (pow2)  $n = 2^k - 1$ 
  by linarith
  then show ?case
  proof cases
    case st-interv
    then show ?thesis apply – apply (rule exI[of - k])
      by (metis (no-types, lifting) One-nat-def Suc-diff-Suc Suc-lessI
         $\langle 2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1 \rangle$  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 $i = (2 :: 'a)^k - (1 :: 'a)$

- *luby-sequence-core* $(i - 2^k - 1 + 1)$, if $(2::'a)^k - 1 \leq i$ and $i \leq (2::'a)^k - (1::'a)$

Then the sequence is then scaled by a constant unit run (called *ur* here), strictly positive.

function *luby-sequence-core* :: *nat* \Rightarrow *nat* **where**

luby-sequence-core *i* =

(if $\exists k. i = 2^k - 1$

then $2^((SOME k. i = 2^k - 1) - 1)$

else *luby-sequence-core* $(i - 2^((SOME k. 2^{(k-1)} \leq i \wedge i < 2^k - 1) - 1) + 1))$)

by *auto*

termination

proof (*relation less-than*, *goal-cases*)

case 1

then show ?*case* **by** *auto*

next

case (2 *i*)

let ?*k* = (*SOME* *k*. $2^k - 1 \leq i \wedge i < 2^{k+1} - 1$)

have $2^{(?k - 1)} \leq i \wedge i < 2^{?k} - 1$

apply (*rule someI-ex*)

using 2 *exists-luby-decomp* **by** *blast*

then show ?*case*

proof –

have $\forall n na. \neg (1::nat) \leq n \vee 1 \leq n \wedge na$

by (*meson one-le-power*)

then have *f1*: $(1::nat) \leq 2^{(?k - 1)}$

using *one-le-numeral* **by** *blast*

have *f2*: $i - 2^{(?k - 1)} + 2^{(?k - 1)} = i$

using $(2^{(?k - 1)} \leq i \wedge i < 2^{?k} - 1)$ *le-add-diff-inverse2* **by** *blast*

have *f3*: $2^{?k} - 1 \neq \text{Suc } 0$

using *f1* $(2^{(?k - 1)} \leq i \wedge i < 2^{?k} - 1)$ **by** *linarith*

have $2^{?k} - (1::nat) \neq 0$

using $(2^{(?k - 1)} \leq i \wedge i < 2^{?k} - 1)$ *gr-implies-not0* **by** *blast*

then have *f4*: $2^{?k} \neq (1::nat)$

by *linarith*

have *f5*: $\forall n na. \text{if } na = 0 \text{ then } (n::nat) \wedge na = 1 \text{ else } n \wedge na = n * n \wedge (na - 1)$

by (*simp add: power-eq-if*)

then have ?*k* $\neq 0$

using *f4* **by** *meson*

then have $2^{(?k - 1)} \neq \text{Suc } 0$

using *f5 f3* **by** *presburger*

then have $\text{Suc } 0 < 2^{(?k - 1)}$

using *f1* **by** *linarith*

then show ?*thesis*

using *f2 less-than-iff* **by** *presburger*

qed

qed

function *natlog2* :: *nat* \Rightarrow *nat* **where**

natlog2 *n* = (if *n* = 0 then 0 else 1 + *natlog2* (*n* div 2))

using *not0-implies-Suc* **by** *auto*

termination **by** (*relation measure* ($\lambda n. n$)) *auto*

declare *natlog2.simps*[*simp del*]

declare *luby-sequence-core.simps*[*simp del*]

lemma *two-pover-n-eq-two-power-n'-eq*:
 assumes $H: (2::nat) \wedge (k::nat) - 1 = 2 \wedge k' - 1$
 shows $k' = k$
proof –
 have $(2::nat) \wedge (k::nat) = 2 \wedge k'$
 using H by (metis *One-nat-def Suc-pred zero-less-numeral zero-less-power*)
 then show *?thesis* by simp
qed

lemma *luby-sequence-core-two-power-minus-one*:
 luby-sequence-core $(2^k - 1) = 2^{(k-1)}$ (is $?L = ?K$)
proof –
 have *decomp*: $\exists ka. 2^k - 1 = 2^{ka} - 1$
 by auto
 have $?L = 2^{((SOME\ k'. (2::nat) \wedge k - 1 = 2^{k'} - 1) - 1)}$
 apply (subst luby-sequence-core.simps, subst *decomp*)
 by simp
 moreover have $(SOME\ k'. (2::nat) \wedge k - 1 = 2^{k'} - 1) = k$
 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
 $H: 2 \wedge (k - Suc\ 0) \leq i$ and
 $k': i < 2 \wedge k' - Suc\ 0$ and
 $k-k': k > k'$
 shows *False*
proof –
 have $2 \wedge k' - Suc\ 0 < 2 \wedge (k - Suc\ 0)$
 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
 $i-k: i < 2^k - 1$
 shows luby-sequence-core $i = luby-sequence-core\ (i - 2 \wedge (k - 1) + 1)$
proof –
 have $H: \neg (\exists ka. i = 2^{ka} - 1)$
 proof (rule ccontr)
 assume $\neg ?thesis$
 then obtain $k': nat$ where $k': i = 2^{k'} - 1$ by blast
 have $(2::nat) \wedge k' - 1 < 2^k - 1$
 using $i-k$ unfolding k' .
 then have $(2::nat) \wedge k' < 2^k$
 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  $(2^{b+1} - 1) = 2^b$ 
    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  $2^{b+1} - 1$ ] 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  $n \geq 1$ 
proof (induction n rule: nat-less-induct-case)
  case 0
  then show ?case by (simp add: luby-sequence-core-0)
next
  case (Suc n) note IH = this

```

```

consider
  (interv) k where  $2 \wedge (k - 1) \leq \text{Suc } n$  and  $\text{Suc } n < 2 \wedge k - 1$ 

```

```

| (pow2) k where Suc n = 2 ^ k - Suc 0
using exists-luby-decomp[of Suc n] by auto

then show ?case
proof cases
  case pow2
  show ?thesis
    using luby-sequence-core-two-power-minus-one pow2 by auto
next
  case interv
  have n: Suc n - 2 ^ (k - 1) + 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 +
  conflict-driven-clause-learningW — functions for clauses:
    mset-cls union-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss

  — functions for the conflicting clause:
    mset-ccls union-ccls insert-ccls remove-clit

  — conversion
    ccls-of-cls cls-of-ccls

  — functions for the state:
    — access functions:
      trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
    — changing state:
      cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
      update-conflicting

    — get state:
      init-state
      restart-state
for
  ur :: nat and
  mset-cls:: 'cls ⇒ 'v clause and
  union-cls :: 'cls ⇒ 'cls ⇒ 'cls and
  insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls and
  remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls and

  mset-clss:: 'clss ⇒ 'v clauses and
  union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
  in-clss :: 'cls ⇒ 'clss ⇒ bool and
  insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
  remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and

```

```

mset-ccls :: 'ccls ⇒ 'v clause and
union-ccls :: 'ccls ⇒ 'ccls ⇒ 'ccls and
insert-ccls :: 'v literal ⇒ 'ccls ⇒ 'ccls and
remove-clit :: 'v literal ⇒ 'ccls ⇒ 'ccls and

ccls-of-cl :: 'cls ⇒ 'ccls and
cls-of-ccls :: 'ccls ⇒ 'cls and

trail :: 'st ⇒ ('v, nat, 'v clause) marked-lits and
hd-raw-trail :: 'st ⇒ ('v, nat, 'cls) marked-lit and
raw-init-clss :: 'st ⇒ 'clss and
raw-learned-clss :: 'st ⇒ 'clss and
backtrack-lvl :: 'st ⇒ nat and
raw-conflicting :: 'st ⇒ 'ccls option and

cons-trail :: ('v, nat, 'cls) marked-lit ⇒ 'st ⇒ 'st and
tl-trail :: 'st ⇒ 'st and
add-init-cl :: 'cls ⇒ 'st ⇒ 'st and
add-learned-cl :: 'cls ⇒ 'st ⇒ 'st and
remove-cl :: 'cls ⇒ 'st ⇒ 'st and
update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
update-conflicting :: 'ccls option ⇒ 'st ⇒ 'st and

init-state :: 'clss ⇒ 'st and
restart-state :: 'st ⇒ 'st
begin

sublocale cdclW-restart - - - - - luby-sequence
  apply unfold-locales
  using bounded-luby-sequence by blast

end
end
theory CDCL-WNOT
imports CDCL-NOT CDCL-W-Termination CDCL-W-Merge
begin

```

21 Link between Weidenbach's and NOT's CDCL

21.1 Inclusion of the states

```

declare upt.simps(2)[simp del]

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, 'lvl, 'a) marked-lit list
  ⇒ ('v, unit, unit) marked-lit list where
  convert-trail-from-W ≡ map convert-marked-lit-from-W

lemma lits-of-l-convert-trail-from-W[simp]:
  lits-of-l (convert-trail-from-W M) = lits-of-l 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 image-image*)

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.

consts *dummy-cls* :: 'cls
fun *convert-marked-lit-from-NOT*
 :: ('a, 'e, 'b) *marked-lit* \Rightarrow ('a, nat, 'cls) *marked-lit* **where**
 convert-marked-lit-from-NOT (*Propagated L -*) = *Propagated L dummy-cls* |
 convert-marked-lit-from-NOT (*Marked L -*) = *Marked L 0*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT \equiv *map convert-marked-lit-from-NOT*

lemma *undefined-lit-convert-trail-from-NOT*[*simp*]:
 undefined-lit (*convert-trail-from-NOT* *F*) *L* \longleftrightarrow *undefined-lit* *F* *L*
by (*induction F rule: marked-lit-list-induct*) (*auto simp: defined-lit-map*)

lemma *lits-of-l-convert-trail-from-NOT*:
 lits-of-l (*convert-trail-from-NOT* *F*) = *lits-of-l* *F*
by (*induction F rule: marked-lit-list-induct*) *auto*

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  $state_W \subseteq dpll\text{-}state\text{-}ops$ 
  mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  raw-clauses
   $\lambda L \ S. \text{cons-trail (convert-marked-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 
  by unfold-locales

context  $state_W$ 
begin
lemma convert-marked-lit-from-W-convert-marked-lit-from-NOT[simp]:
  convert-marked-lit-from-W (mmset-of-mlit (convert-marked-lit-from-NOT L)) = L
  by (cases L) auto
end

sublocale  $state_W \subseteq dpll\text{-}state$ 
  mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  raw-clauses
   $\lambda L \ S. \text{cons-trail (convert-marked-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 
  by unfold-locales (auto simp: map-tl o-def)

context  $state_W$ 
begin
declare  $state\text{-}simp_{NOT}[simp \text{ del}]$ 
end

sublocale  $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}ops$ 
  mset-cls union-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
   $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$ 
  raw-clauses
   $\lambda L \ S. \text{cons-trail (convert-marked-lit-from-NOT } L) \ S$ 
   $\lambda S. \text{tl-trail } S$ 
   $\lambda C \ S. \text{add-learned-cls } C \ S$ 
   $\lambda C \ S. \text{remove-cls } C \ S$ 
   $\lambda - . \text{True}$ 
   $\lambda - \ S. \text{raw-conflicting } S = \text{None}$ 
   $\lambda C \ C' \ L' \ S \ T. \text{backjump-l-cond } C \ C' \ L' \ S \ T$ 
   $\wedge \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$ 
  by unfold-locales

thm  $cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}proxy.axioms$ 
sublocale  $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}proxy$ 
  mset-cls union-cls insert-cls remove-lit

```

```

mset-clss union-clss in-clss insert-clss remove-from-clss
λS. convert-trail-from-W (trail S)
raw-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
λ- S. raw-conflicting S = None
backjump-l-cond
invNOT
proof (unfold-locales, goal-cases)
  case 2
  then show ?case using cdclNOT-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 ∉ # C'
      using ⟨F ⊨as CNot C'⟩ ⟨undefined-lit F L⟩ Marked-Propagated-in-iff-in-lits-of-l
      in-CNot-implies-uminus(2) by fast
    then have distinct-mset (?C' + {#L#})
      by (simp add: distinct-mset-single-add)
  moreover
    have no-dup F
      using ⟨invNOT S⟩ ⟨convert-trail-from-W (trail S) = F' @ Marked K () # F⟩
      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-l F)
      using distinct-consistent-interp by blast
    then have ¬ tautology (C')
      using ⟨F ⊨as CNot C'⟩ consistent-CNot-not-tautology true-annots-true-cls by blast
    then have ¬ tautology (?C' + {#L#})
      using ⟨F ⊨as CNot C'⟩ ⟨undefined-lit F L⟩ by (metis CNot-remdups-mset
        Marked-Propagated-in-iff-in-lits-of-l 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 ⟨invNOT S⟩ unfolding invNOT-def by (simp add: o-def)
    have f3: atm-of L ∈ atms-of-mm (clauses S)
      ∪ atm-of ' lits-of-l (convert-trail-from-W (trail S))
      using ⟨convert-trail-from-W (trail S) = F' @ Marked K () # F⟩
      ⟨atm-of L ∈ atms-of-mm (clauses S) ∪ atm-of ' lits-of-l (F' @ Marked K () # F)⟩ by auto
    have f4: clauses S ⊨pm remdups-mset C' + {#L#}
      by (metis (no-types) ⟨L ∉ # C'⟩ ⟨clauses S ⊨pm C' + {#L#}⟩ remdups-mset-singleton-sum(2)
        true-clss-cls-remdups-mset union-commute)
    have F ⊨as CNot (remdups-mset C')
      by (simp add: ⟨F ⊨as CNot C'⟩)
    obtain D where D: mset-cls D = remdups-mset C' + {#L#}
      using ex-mset-cls by blast
    have Ex (backjump-l S)
      apply standard

```

```

    apply (rule backjump-l.intros[OF - f2, of - - - ])
    using f4 f3 f2 <¬ tautology (remdups-mset C' + {#L#})>
    calculation(2-5,9) <F ⊨as CNot (remdups-mset C')>
    state-eqNOT-ref D unfolding backjump-l-cond-def by blast+
  then show ?thesis
    by blast
qed
qed

sublocale conflict-driven-clause-learningW ⊆ cdclNOT-merge-bj-learn-proxy2 - - - - -
  λS. convert-trail-from-W (trail S)
  raw-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
  λ-. S. raw-conflicting S = None backjump-l-cond invNOT
by unfold-locales

sublocale conflict-driven-clause-learningW ⊆ cdclNOT-merge-bj-learn - - - - -
  λS. convert-trail-from-W (trail S)
  raw-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
  backjump-l-cond
  λ-. True
  λ-. S. raw-conflicting S = None invNOT
apply unfold-locales
  using dpll-bj-no-dup apply (simp add: comp-def)
  using cdclNOT.simps cdclNOT-no-dup no-dup-convert-from-W unfolding invNOT-def by blast

context conflict-driven-clause-learningW
begin

Notations are lost while proving locale inclusion:
notation state-eqNOT (infix ~NOT 50)

```

21.2 Additional Lemmas between NOT and W states

lemma trail_W-eq-reduce-trail-to_{NOT}-eq:
 trail S = trail T \implies trail (reduce-trail-to_{NOT} F S) = trail (reduce-trail-to_{NOT} F T)

proof (induction F S arbitrary: T rule: reduce-trail-to_{NOT}.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-to_{NOT} F (tl-trail S)) = trail (reduce-trail-to_{NOT} F (tl-trail T))
using IH **by** (metis (no-types) trail-tl-trail)
then show trail (reduce-trail-to_{NOT} F S) = trail (reduce-trail-to_{NOT} F T)
using tr **by** (metis (no-types) reduce-trail-to_{NOT}.elim)
 qed

lemma trail-reduce-trail-to_{NOT}-add-learned-cls:
 no-dup (trail S) \implies

trail (*reduce-trail-to*_{NOT} *M* (*add-learned-cls* *D S*)) = *trail* (*reduce-trail-to*_{NOT} *M S*)
by (*rule* *trail_W-eq-reduce-trail-to_{NOT}-eq*) *simp*

lemma *reduce-trail-to*_{NOT}-*reduce-trail-convert*:
*reduce-trail-to*_{NOT} *C S* = *reduce-trail-to* (*convert-trail-from-NOT* *C*) *S*
apply (*induction* *C S* *rule*: *reduce-trail-to*_{NOT}.*induct*)
apply (*subst* *reduce-trail-to*_{NOT}.*simps*, *subst* *reduce-trail-to*.*simps*)
by *auto*

lemma *reduce-trail-to-map*[*simp*]:
reduce-trail-to (*map* *f M*) *S* = *reduce-trail-to* *M S*
by (*rule* *reduce-trail-to-length*) *simp*

lemma *reduce-trail-to*_{NOT}-*length*:
length *M* = *length* *M'* \implies *reduce-trail-to*_{NOT} *M S* = *reduce-trail-to*_{NOT} *M' S*
apply (*induction* *M S* *arbitrary*: *rule*: *reduce-trail-to*_{NOT}.*induct*)
by (*simp* *add*: *reduce-trail-to*_{NOT}.*simps*)

lemma *reduce-trail-to*_{NOT}-*map*[*simp*]:
*reduce-trail-to*_{NOT} (*map* *f M*) *S* = *reduce-trail-to*_{NOT} *M S*
by (*rule* *reduce-trail-to*_{NOT}-*length*) *simp*

lemma *skip-or-resolve-state-change*:
assumes *skip-or-resolve*** *S T*
shows
 $\exists M. \text{trail } S = M @ \text{trail } T \wedge (\forall m \in \text{set } M. \neg \text{is-marked } m)$
clauses *S* = *clauses* *T*
backtrack-lvl *S* = *backtrack-lvl* *T*
using *assms*

proof (*induction* *rule*: *rtranclp-induct*)
case *base*
case 1 **show** ?*case* **by** *simp*
case 2 **show** ?*case* **by** *simp*
case 3 **show** ?*case* **by** *simp*

next
case (*step* *T U*) **note** *st* = *this*(1) **and** *s-o-r* = *this*(2) **and** *IH* = *this*(3) **and** *IH'* = *this*(3-5)

case 2 **show** ?*case* **using** *IH'* *s-o-r* **by** (*auto* *elim*!: *rulesE* *simp*: *skip-or-resolve.simps*)
case 3 **show** ?*case* **using** *IH'* *s-o-r* **by** (*auto* *elim*!: *rulesE* *simp*: *skip-or-resolve.simps*)

case 1 **show** ?*case*
using *s-o-r*
proof *cases*
case *s-or-r-skip*
then **show** ?*thesis* **using** *IH* **by** (*auto* *elim*!: *rulesE* *simp*: *skip-or-resolve.simps*)
next
case *s-or-r-resolve*
then **show** ?*thesis*
using *IH* **by** (*cases* *trail T*) (*auto* *elim*!: *rulesE* *simp*: *skip-or-resolve.simps* *dest*!:
hd-raw-trail)

qed
qed

21.3 More lemmas conflict-propagate and backjumping

21.4 CDCL FW

lemma *cdcl_W-merge-is-cdcl_{NOT}-merged-bj-learn*:

assumes

inv: *cdcl_W-all-struct-inv S* **and**

cdcl_W:cdcl_W-merge S T

shows *cdcl_{NOT}-merged-bj-learn S T*

\vee (*no-step cdcl_W-merge T* \wedge *conflicting T* \neq *None*)

using *cdcl_W inv*

proof *induction*

case (*fw-propagate S T*) **note** *propa = this(1)*

then obtain *M N U k L C* **where**

H: *state S = (M, N, U, k, None)* **and**

CL: *C + {#L#} ∈ # clauses S* **and**

M-C: *M ⊨_{as} CNot C* **and**

undef: *undefined-lit (trail S) L* **and**

T: *state T = (Propagated L (C + {#L#}) # M, N, U, k, None)*

by (*auto elim: propagate-high-levelE*)

have *propagate_{NOT} S T*

using *H CL T undef M-C* **by** (*auto simp: state-eq_{NOT}-def state-eq-def raw-clauses-def*
simp del: state-simp)

then show *?case*

using *cdcl_{NOT}-merged-bj-learn.intros(2)* **by** *blast*

next

case (*fw-decide S T*) **note** *dec = this(1)* **and** *inv = this(2)*

then obtain *L* **where**

undef-L: *undefined-lit (trail S) L* **and**

atm-L: *atm-of L ∈ atms-of-mm (init-clss S)* **and**

T: *T ∼ cons-trail (Marked L (Suc (backtrack-lvl S)))*

(update-backtrack-lvl (Suc (backtrack-lvl S)) S)

by (*auto elim: decideE*)

have *decide_{NOT} S T*

apply (*rule decide_{NOT}.decide_{NOT}*)

using *undef-L apply simp*

using *atm-L inv unfolding cdcl_W-all-struct-inv-def no-strange-atm-def raw-clauses-def*

apply *auto[]*

using *T undef-L unfolding state-eq-def state-eq_{NOT}-def* **by** (*auto simp: raw-clauses-def*)

then show *?case* **using** *cdcl_{NOT}-merged-bj-learn-decide_{NOT}* **by** *blast*

next

case (*fw-forget S T*) **note** *rf = this(1)* **and** *inv = this(2)*

then obtain *C* **where**

S: *conflicting S = None* **and**

C-le: *C !∈ ! raw-learned-clss S* **and**

\neg (*trail S*) \models_{asm} *clauses S* **and**

mset-cls C \notin *set (get-all-mark-of-propagated (trail S))* **and**

C-init: *mset-cls C* \notin *init-clss S* **and**

T: *T ∼ remove-cls C S*

by (*auto elim: forgetE*)

have *init-clss S ⊨_{pm} mset-cls C*

using *inv C-le unfolding cdcl_W-all-struct-inv-def cdcl_W-learned-clause-def raw-clauses-def*

by (*meson in-clss-mset-clss true-clss-clss-in-imp-true-clss-clss*)

then have *S-C*: *removeAll-mset (mset-cls C) (clauses S) ⊨_{pm} mset-cls C*

using *C-init C-le unfolding raw-clauses-def* **by** (*auto simp add: Un-Diff ac-simps*)

have *forget_{NOT} S T*

```

apply (rule forgetNOT.forgetNOT)
  using S-C apply blast
  using S apply simp
  using C-init C-le apply (simp add: raw-clauses-def)
using T C-le C-init by (auto
  simp: state-eq-def Un-Diff state-eqNOT-def raw-clauses-def ac-simps
  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 CT where
  confl-T: raw-conflicting T = Some CT and
  CT: mset-ccls CT = mset-cls CS and
  CS: CS !∈! raw-clauses S and
  tr-S-CS: trail S ⊨as CNot (mset-cls CS)
  using confl by (elim conflictE) (auto simp del: state-simp simp: state-eq-def)
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 rtrancpl-cdclW-bj-skip-or-resolve-backtrack unfolding full-def by meson
then show ?case
proof cases
  case no-bt
  then have conflicting U ≠ None
  using confl by (induction rule: rtrancpl-induct)
  (auto simp del: state-simp simp: skip-or-resolve.simps state-eq-def elim!: rulesE)
  moreover then have no-step cdclW-merge U
  by (auto simp: cdclW-merge.simps elim: rulesE)
  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-rtrancpl[of skip-or-resolve cdclW] rtrancpl-skip-or-resolve-rtrancpl-cdclW
  by blast
then have cdclW-M-level-inv T'
  using rtrancpl-cdclW-consistent-inv (cdclW-M-level-inv T) by blast
then obtain M1 M2 i D L K where
  confl-T': raw-conflicting T' = Some D and
  LD: L ∈# mset-ccls D and
  M1-M2: (Marked K (i+1) # M1, M2) ∈ set (get-all-marked-decomposition (trail T')) and
  get-level (trail T') L = backtrack-lvl T' and
  get-level (trail T') L = get-maximum-level (trail T') (mset-ccls D) and
  get-maximum-level (trail T') (mset-ccls (remove-clit L D)) = i and
  undef-L: undefined-lit M1 L and
  U: U ∼ cons-trail (Propagated L (cls-of-ccls D))
  (reduce-trail-to M1
  (add-learned-cls (cls-of-ccls D)
  (update-backtrack-lvl i
  (update-conflicting None T'))))))
  using bt by (auto elim: backtrack-levE)
have [simp]: clauses S = clauses T

```

```

    using confl by (auto elim: rulesE)
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 simp: skip-or-resolve.simps elim: rulesE)
      then show ?case using IH by auto
    qed
have inv-T: cdclW-all-struct-inv T
  by (meson cdclW-cp.simps confl inv r-into-rtrancpl rtrancpl-cdclW-all-struct-inv-inv
      rtrancpl-cdclW-cp-rtrancpl-cdclW)
have cdclW** T T'
  using rtrancpl-skip-or-resolve-rtrancpl-cdclW s-or-r by blast
have inv-T': cdclW-all-struct-inv T'
  using ⟨cdclW** T T'⟩ inv-T rtrancpl-cdclW-all-struct-inv-inv by blast
have inv-U: cdclW-all-struct-inv U
  using cdclW-merge-restart-cdclW confl fw-r-conflict inv local.bj
  rtrancpl-cdclW-all-struct-inv-inv by blast

have [simp]: init-clss S = init-clss T'
  using ⟨cdclW** T T'⟩ cdclW-init-clss confl cdclW-all-struct-inv-def conflict inv
  by (metis ⟨cdclW-M-level-inv T'⟩ rtrancpl-cdclW-init-clss)
then have atm-L: atm-of L ∈ atms-of-mm (clauses S)
  using inv-T' confl-T' LD unfolding cdclW-all-struct-inv-def no-strange-atm-def
  raw-clauses-def
  by (simp add: atms-of-def image-subset-iff)
obtain M where tr-T: trail T = M @ trail T'
  using s-or-r skip-or-resolve-state-change by meson
obtain M' where
  tr-T': trail T' = M' @ Marked K (i+1) # tl (trail U) and
  tr-U: trail U = Propagated L (mset-ccls D) # 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 elim: rulesE)
have init-clss T' + learned-clss S ⊨pm mset-ccls D
  using inv-T' confl-T' unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def
  raw-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 M1 S) = M1
  using M1-M2 confl by (subst reduce-trail-toNOT-reduce-trail-convert)
  (auto simp: comp-def elim: rulesE)

```



```

have every-mark-is-a-conflict  $U$ 
  using inv- $U$  unfolding cdcl $_W$ -all-struct-inv-def cdcl $_W$ -conflicting-def by simp
then have  $U-D$ :  $tl\ (trail\ U) \models_{as} CNot\ (remove1-mset\ L\ (mset-ccls\ D))$ 
  by (metis append-self-conv2 tr- $U$ )
thm backjump-l[of - - - -  $L$  cls-of-ccls  $D$  - remove1-mset  $L$  (mset-ccls  $D$ )]
have backjump-l  $S\ U$ 
  apply (rule backjump-l[of - - - -  $L$  cls-of-ccls  $D$  - remove1-mset  $L$  (mset-ccls  $D$ )])
    using tr- $T$  apply simp
    using inv unfolding cdcl $_W$ -all-struct-inv-def cdcl $_W$ - $M$ -level-inv-def
    apply (simp add: comp-def)
  using  $U\ M1-M2\ confl\ undef-L\ M1-M2\ inv-T'\ inv\ undef-L$  unfolding cdcl $_W$ -all-struct-inv-def
    cdcl $_W$ - $M$ -level-inv-def apply (auto simp: state-eq $_{NOT}$ -def
      trail-reduce-trail-to $_{NOT}$ -add-learned-clcs)[]
  using  $C_S$  apply auto[]
  using tr- $S$ - $C_S$  apply simp

  using  $U\ undef-L\ M1-M2\ inv-T'\ inv$  unfolding cdcl $_W$ -all-struct-inv-def
    cdcl $_W$ - $M$ -level-inv-def apply auto[]
  using undef- $L\ atm-L$  apply (simp add: trail-reduce-trail-to $_{NOT}$ -add-learned-clcs)
  using (init-clcs  $T' + learned-clcs\ S \models_{pm} mset-ccls\ D$ )  $LD$  unfolding raw-clauses-def
  apply simp
  using  $LD$  apply simp
  apply (metis  $U-D$  convert-trail-from- $W$ -true-annots)
  using inv- $T'\ inv-U\ U\ confl-T'\ undef-L\ M1-M2\ LD$  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**  $S\ T \vee (no-step\ cdcl_W-merge\ T \wedge conflicting\ T \neq None)$ 
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  $S\ T \vee (no-step\ cdcl_W-merge\ T \wedge conflicting\ T \neq None)$ 
    using inv cdcl $_W$ -merge-is-cdcl $_{NOT}$ -merged-bj-learn by blast
  have inv $S$ : 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 inv $S$  by (simp add: comp-def)
  have cdcl $_{NOT} \leq cdcl_{NOT}$ -restart

```

```

    by (auto simp: restart-ops.cdclNOT-raw-restart.simps)
  then show ?thesis
    using ff3 ff2 IH cdclNOT-merged-bj-learn-is-tranclp-cdclNOT
      rtranclp-mono[of cdclNOT cdclNOT-restart] invS predicate2D by blast
next
  case fw-r
  then show ?thesis by (blast intro: restart-ops.cdclNOT-raw-restart.intros)
qed
qed

abbreviation  $\mu_{FW} :: 'st \Rightarrow nat$  where
 $\mu_{FW} S \equiv (if\ no\_step\ cdcl_W\ merge\ S\ then\ 0\ else\ 1 + \mu_{CDCL}'\ merged\ (set\_mset\ (init\_clss\ S))\ S)$ 

lemma cdclW-merge- $\mu_{FW}$ -decreasing:
assumes
  inv: cdclW-all-struct-inv S and
  fw: cdclW-merge S T
shows  $\mu_{FW} T < \mu_{FW} S$ 
proof -
  let ?A = init-clss S
  have atm-clauses: atms-of-mm (clauses S)  $\subseteq$  atms-of-mm ?A
    using inv unfolding cdclW-all-struct-inv-def no-strange-atm-def raw-clauses-def by auto
  have atm-trail: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-mm ?A
    using inv unfolding cdclW-all-struct-inv-def no-strange-atm-def raw-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]:  $\neg no\_step\ cdcl_W\ merge\ S$ 
    using fw by auto
  have [simp]: init-clss S = init-clss T
    using cdclW-merge-restart-cdclW[of S T] inv rtranclp-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, of T] atm-trail n-d
        by (auto split: if-split simp: comp-def image-image)
    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

sublocale conflict-driven-clause-learningW-termination
by unfold-locals (simp add: wf-cdclW-merge)

```

lemma *full-cdcl_W-s'-full-cdcl_W-merge-restart*:

assumes

conflicting $R = \text{None}$ **and**

inv: *cdcl_W-all-struct-inv* R

shows *full cdcl_W-s' R V* \longleftrightarrow *full cdcl_W-merge-stgy R V* (**is** $?s' \longleftrightarrow ?fw$)

proof

assume $?s'$

then have *cdcl_W-s'^** R V* **unfolding** *full-def* **by** *blast*

have *cdcl_W-all-struct-inv V*

using $\langle \text{cdcl}_W\text{-s}'^{**} R V \rangle$ *inv rtranclp-cdcl_W-all-struct-inv-inv rtranclp-cdcl_W-s'-rtranclp-cdcl_W*

by *blast*

then have *n-s: no-step cdcl_W-merge-stgy V*

using *no-step-cdcl_W-s'-no-step-cdcl_W-merge-stgy* **by** (*meson* $\langle \text{full cdcl}_W\text{-s}' R V \rangle$ *full-def*)

have *n-s-bj: no-step cdcl_W-bj V*

by (*metis* $\langle \text{cdcl}_W\text{-all-struct-inv } V \rangle \langle \text{full cdcl}_W\text{-s}' R V \rangle$ *bj full-def*

n-step-cdcl_W-stgy-iff-no-step-cdcl_W-cl-cdcl_W-o)

have *n-s-cp: no-step cdcl_W-merge-cp V*

proof –

 { **fix** $ss :: 'st$

obtain $ssa :: 'st \Rightarrow 'st$ **where**

$\text{ff1} : \forall s. \neg \text{cdcl}_W\text{-all-struct-inv } s \vee \text{cdcl}_W\text{-s'-without-decide } s (ssa \ s)$

$\vee \text{no-step cdcl}_W\text{-merge-cp } s$

using *conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp* **by** *moura*

have $(\forall p \ s \ sa. \neg \text{full } p (s::'st) \ sa \vee p^{**} \ s \ sa \wedge \text{no-step } p \ sa)$ **and**

$(\forall p \ s \ sa. (\neg p^{**} (s::'st) \ sa \vee (\exists s. p \ sa \ s))) \vee \text{full } p \ s \ sa)$

by (*meson full-def*) +

then have $\neg \text{cdcl}_W\text{-merge-cp } V \ ss$

using ff1 **by** (*metis* (*no-types*) $\langle \text{cdcl}_W\text{-all-struct-inv } V \rangle \langle \text{full cdcl}_W\text{-s}' R V \rangle$ *cdcl_W-s'.simps*

cdcl_W-s'-without-decide.cases) }

then show $?thesis$

by *blast*

qed

consider

 (*fw-no-confl*) *cdcl_W-merge-stgy** R V* **and** *conflicting V = None*

 | (*fw-confl*) *cdcl_W-merge-stgy** R V* **and** *conflicting V ≠ None* **and** *no-step cdcl_W-bj V*

 | (*fw-dec-confl*) $S \ T \ U$ **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*

 | (*fw-dec-no-confl*) $S \ T$ **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 = None*

 | (*cp-no-confl*) *cdcl_W-merge-cp** R V* **and** *conflicting V = None*

 | (*cp-confl*) U **where** *cdcl_W-merge-cp** R U* **and** *conflict U V*

using *rtranclp-cdcl_W-s'-no-step-cdcl_W-s'-without-decide-decomp-into-cdcl_W-merge*[*OF*

$\langle \text{cdcl}_W\text{-s}'^{**} R V \rangle$ *assms*] **by** *auto*

then show $?fw$

proof *cases*

case *fw-no-confl*

then show $?thesis$ **using** *n-s* **unfolding** *full-def* **by** *blast*

next

case *fw-confl*

then show $?thesis$ **using** *n-s* **unfolding** *full-def* **by** *blast*

next

case *fw-dec-confl*

have *cdcl_W-merge-cp U V*

using *n-s-bj* **by** (*metis* *cdcl_W-merge-cp.simps full-unfold fw-dec-confl(5)*)

then have *full1 cdcl_W-merge-cp T V*

```

    unfolding full1-def by (metis fw-dec-confl(4) n-s-cp tranclp-unfold-end)
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case fw-dec-no-confl
  then have full cdclW-merge-cp T V
    using n-s-cp unfolding full-def by blast
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case cp-no-confl
  then have full cdclW-merge-cp R V
    by (simp add: full-def n-s-cp)
  then have R = V ∨ cdclW-merge-stgy++ R V
    using fw-s-cp unfolding full-unfold fw-s-cp
    by (metis (no-types) rtranclp-unfold tranclp-unfold-end)
  then show ?thesis
    by (simp add: full-def n-s rtranclp-unfold)
next
  case cp-confl
  have full cdclW-bj V V
    using n-s-bj unfolding full-def by blast
  then have full1 cdclW-merge-cp R V
    unfolding full1-def by (meson cdclW-merge-cp.conflict' cp-confl(1,2) n-s-cp
      rtranclp-into-tranclp1)
  then show ?thesis using n-s unfolding full-def by auto
qed
next
  assume ?fw
  then have cdclW** R V using rtranclp-mono[of cdclW-merge-stgy cdclW**]
    cdclW-merge-stgy-rtranclp-cdclW unfolding full-def by auto
  then have inv': cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
  have cdclW-s*** R V
    using ⟨?fw⟩ by (simp add: full-def inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s')
  moreover have no-step cdclW-s' V
  proof cases
    assume conflicting V = None
    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 ≠ None
    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 elim: rulesE)
  qed
  ultimately show ?s' unfolding full-def by blast
qed

lemma full-cdclW-stgy-full-cdclW-merge:
  assumes
    conflicting R = None and
    inv: cdclW-all-struct-inv R

```

```

shows  $full\ cdcl_W\text{-}stgy\ R\ V \longleftrightarrow full\ cdcl_W\text{-}merge\text{-}stgy\ R\ V$ 
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 (mset-clss N)
  shows (conflicting S' = Some {#}  $\wedge$  unsatisfiable (set-mset (mset-clss N))
     $\vee$  (conflicting S' = None  $\wedge$  trail S'  $\models_{asm}$  mset-clss N  $\wedge$  satisfiable (set-mset (mset-clss 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) = None
    by auto
  ultimately show ?thesis
    using full full-cdclW-stgy-final-state-conclusive-from-init-state
    full-cdclW-stgy-full-cdclW-merge no-d by presburger
qed
end

end
theory CDCL-W-Incremental
imports CDCL-W-Termination
begin

```

22 Incremental SAT solving

```

context conflict-driven-clause-learningW
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  $\longleftrightarrow$ 
    conflict-is-false-with-level S
     $\wedge$  no-clause-is-false S
     $\wedge$  no-smaller-confl S
     $\wedge$  no-clause-is-false S

```

```

lemma cdclW-stgy-cdclW-stgy-invariant:
  assumes
    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 (intro conjI)
  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]
  using cdclW cdclW-stgy-not-non-negated-init-clss apply simp
  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
*cdcl_W: cdcl_W-stgy** S T and*
inv-s: cdcl_W-stgy-invariant S and
inv: cdcl_W-all-struct-inv S
shows
cdcl_W-stgy-invariant T
using *assms* **apply** (*induction*)
apply *simp*
using *cdcl_W-stgy-cdcl_W-stgy-invariant rtrancp-cdcl_W-all-struct-inv-inv*
rtrancp-cdcl_W-stgy-rtrancp-cdcl_W **by** *blast*

abbreviation *decr-bt-lvl* **where**
decr-bt-lvl S \equiv *update-backtrack-lvl (backtrack-lvl S - 1) S*

When we add a new clause, we reduce the trail until we get to the first literal included in C. Then we can mark the conflict.

fun *cut-trail-wrt-clause* **where**
cut-trail-wrt-clause C [] S = S |
cut-trail-wrt-clause C (Marked L - # M) S =
(if -L \in # C then S
else cut-trail-wrt-clause C M (decr-bt-lvl (tl-trail S))) |
cut-trail-wrt-clause C (Propagated L - # M) S =
(if -L \in # C then S
else cut-trail-wrt-clause C M (tl-trail S))

definition *add-new-clause-and-update* :: *'ccls \Rightarrow 'st \Rightarrow 'st* **where**
add-new-clause-and-update C S =
(if trail S \models as CNot (mset-ccls C)
then update-conflicting (Some C) (add-init-cls (cls-of-ccls C)
(cut-trail-wrt-clause (mset-ccls C) (trail S) S))
else add-init-cls (cls-of-ccls 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)
  show ?case
  proof (cases count C (-L) = 0)
    case False
    then show ?thesis
      using IH M bt by (auto simp: true-annots-true-cls)
  next
    case True
    obtain mma :: 'v literal multiset where
      f6: (mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow M \models_a mma\} \longrightarrow M \models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ )
      using true-annots-def by blast
    have mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow \text{trail } T \models_a mma$ 
      using CNot-def M bt by (metis (no-types) true-annots-def)
    then have  $M \models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ 
      using f6 True M bt by (force simp: count-eq-zero-iff)
    then show ?thesis
      using IH true-annots-true-cls M by (auto simp: CNot-def)
  qed
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt = this(3)
  show ?case
  proof (cases count C (-L) = 0)
    case False
    then show ?thesis
      using IH M bt by (auto simp: true-annots-true-cls)
  next
    case True
    obtain mma :: 'v literal multiset where
      f6: (mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow M \models_a mma\} \longrightarrow M \models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ )
      using true-annots-def by blast
    have mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow \text{trail } T \models_a mma$ 
      using CNot-def M bt by (metis (no-types) true-annots-def)
    then have  $M \models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ 
      using f6 True M bt by (force simp: count-eq-zero-iff)
    then show ?thesis
      using IH true-annots-true-cls M by (auto simp: CNot-def)
  qed
qed

lemma cut-trail-wrt-clause-hd-trail-in-or-empty-trail:
  (( $\forall L \in \# C. -L \notin \text{lits-of-}l(\text{trail } T)$ )  $\wedge$  trail (cut-trail-wrt-clause C (trail T) T) = [])
   $\vee$  ( $-\text{lit-of}(\text{hd}(\text{trail}(\text{cut-trail-wrt-clause } C(\text{trail } T) T))) \in \# C$ 
     $\wedge$  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 $incremental-cdcl_W :: 'st \Rightarrow 'st \Rightarrow bool$ **for** S **where**

add-conflict:

```

trail S  $\models_{asm}$  init-clss S  $\implies$  distinct-mset (mset-ccls C)  $\implies$  conflicting S = None  $\implies$ 
trail S  $\models_{as}$  CNot (mset-ccls C)  $\implies$ 
full cdclW-stgy
  (update-conflicting (Some C)
    (add-init-cls (cls-of-ccls C) (cut-trail-wrt-clause (mset-ccls C) (trail S) S))) T  $\implies$ 
incremental-cdclW S T |

```

add-no-conflict:

```

trail S  $\models_{asm}$  init-clss S  $\implies$  distinct-mset (mset-ccls C)  $\implies$  conflicting S = None  $\implies$ 
 $\neg$ trail S  $\models_{as}$  CNot (mset-ccls C)  $\implies$ 
full cdclW-stgy (add-init-cls (cls-of-ccls C) S) T  $\implies$ 
incremental-cdclW S T

```

lemma $cdcl_W$ -all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv:

assumes

```

inv-T: cdclW-all-struct-inv T and
tr-T-N[simp]: trail T  $\models_{asm}$  N and
tr-C[simp]: trail T  $\models_{as}$  CNot (mset-ccls C) and
[simp]: distinct-mset (mset-ccls C)

```

shows $cdcl_W$ -all-struct-inv (add-new-clause-and-update C T) (is cdcl_W-all-struct-inv ?T')

proof –

```

let ?T = update-conflicting (Some C)
  (add-init-cls (cls-of-ccls C) (cut-trail-wrt-clause (mset-ccls C) (trail T) T))

```

obtain M **where**

```

M: trail T = M @ trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)
using trail-cut-trail-wrt-clause[of T mset-ccls C] by blast

```

have $H[dest]: \bigwedge x. x \in lits-of-l$ (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)) \implies
 $x \in lits-of-l$ (trail T)

using inv-T arg-cong[OF M, of lits-of-l] **by** auto

have $H'[dest]: \bigwedge x. x \in set$ (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)) \implies
 $x \in set$ (trail T)

using inv-T arg-cong[OF M, of set] **by** auto

have H -proped: $\bigwedge x. x \in set$ (get-all-mark-of-propagated (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T))) \implies $x \in set$ (get-all-mark-of-propagated (trail T))

using inv-T arg-cong[OF M, of get-all-mark-of-propagated] **by** auto

have [simp]: no-strange-atm ?T

using inv-T **unfolding** $cdcl_W$ -all-struct-inv-def no-strange-atm-def add-new-clause-and-update-def
 $cdcl_W$ -M-level-inv-def **by** (auto 20 1)

have M -lev: $cdcl_W$ -M-level-inv T

using inv-T **unfolding** $cdcl_W$ -all-struct-inv-def **by** blast

then have no-dup (M @ trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T))

unfolding $cdcl_W$ -M-level-inv-def **unfolding** M[symmetric] **by** auto

```

then have [simp]: no-dup (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T))
  by auto

have consistent-interp (lits-of-l (M @ trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)))
  using M-lev unfolding cdclW-M-level-inv-def unfolding M[symmetric] by auto
then have [simp]: consistent-interp (lits-of-l (trail (cut-trail-wrt-clause (mset-ccls C)
  (trail T) T)))
  unfolding consistent-interp-def by auto

have [simp]: cdclW-M-level-inv ?T
  using M-lev cut-trail-wrt-clause-get-all-levels-of-marked[of T mset-ccls C]
  unfolding cdclW-M-level-inv-def by (auto dest: H H'
    simp: M-lev 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  $\models_{as} CNot (mset-ccls C)$ 
  by (simp add: cut-trail-wrt-clause-CNot-trail)
then have [simp]: cdclW-conflicting ?T
  unfolding cdclW-conflicting-def apply simp
  by (metis M  $\langle cdcl_W\text{-conflicting } T \rangle$  append-assoc cdclW-conflicting-decomp(2))

have
  decomp-T: all-decomposition-implies-m (init-clss T) (get-all-marked-decomposition (trail T))
  using inv-T unfolding cdclW-all-struct-inv-def by auto
have all-decomposition-implies-m (init-clss ?T)
  (get-all-marked-decomposition (trail ?T))
  unfolding all-decomposition-implies-def
  proof clarify
    fix a b
    assume (a, b)  $\in$  set (get-all-marked-decomposition (trail ?T))
    from in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend[OF this, of M]
    obtain b' where
      (a, b' @ b)  $\in$  set (get-all-marked-decomposition (trail T))
      using M by auto
    then have unmark-l a  $\cup$  set-mset (init-clss T)  $\models_{ps}$  unmark-l (b' @ b)
      using decomp-T unfolding all-decomposition-implies-def by fastforce
    then have unmark-l a  $\cup$  set-mset (init-clss ?T)  $\models_{ps}$  unmark-l (b @ b')
      by (simp add: Un-commute)
    then show unmark-l a  $\cup$  set-mset (init-clss ?T)  $\models_{ps}$  unmark-l 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: raw-clauses-def)
show ?thesis

```

using $\langle \text{all-decomposition-implies-m } (\text{init-clss } ?T) \text{ (get-all-marked-decomposition (trail } ?T)) \rangle$
unfolding $\text{cdcl}_W\text{-all-struct-inv-def}$ **by** $(\text{auto simp: add-new-clause-and-update-def})$
qed

lemma $\text{cdcl}_W\text{-all-struct-inv-add-new-clause-and-update-cdcl}_W\text{-stgy-inv}$:

assumes

$\text{inv-s: cdcl}_W\text{-stgy-invariant } T$ **and**
 $\text{inv: cdcl}_W\text{-all-struct-inv } T$ **and**
 $\text{tr-T-N[simp]: trail } T \models_{\text{asm}} N$ **and**
 $\text{tr-C[simp]: trail } T \models_{\text{as}} \text{CNot (mset-ccls } C)$ **and**
 $[\text{simp}]: \text{distinct-mset (mset-ccls } C)$

shows $\text{cdcl}_W\text{-stgy-invariant (add-new-clause-and-update } C \text{ } T)$
 $(\text{is cdcl}_W\text{-stgy-invariant } ?T')$

proof –

have $\text{cdcl}_W\text{-all-struct-inv } ?T'$

using $\text{cdcl}_W\text{-all-struct-inv-add-new-clause-and-update-cdcl}_W\text{-all-struct-inv assms}$ **by** blast

then have

$\text{no-dup-cut-T[simp]: no-dup (trail (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T))}$ **and**
 $\text{n-d[simp]: no-dup (trail } T)$

using $\text{cdcl}_W\text{-M-level-inv-decomp(2) cdcl}_W\text{-all-struct-inv-def inv}$
 $\text{n-dup-no-dup-trail-cut-trail-wrt-clause}$ **by** blast+

then have $\text{trail (add-new-clause-and-update } C \text{ } T) \models_{\text{as}} \text{CNot (mset-ccls } C)$
by $(\text{simp add: add-new-clause-and-update-def cut-trail-wrt-clause-CNot-trail cdcl}_W\text{-M-level-inv-def cdcl}_W\text{-all-struct-inv-def})$

obtain MT **where**

$MT: \text{trail } T = MT @ \text{trail (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T)$

using $\text{trail-cut-trail-wrt-clause}$ **by** blast

consider

$(\text{false}) \forall L \in \# \text{mset-ccls } C. - L \notin \text{lits-of-l (trail } T)$ **and**
 $\text{trail (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T) = []$

$| (\text{not-false})$

$- \text{lit-of (hd (trail (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T)))} \in \# (\text{mset-ccls } C)$ **and**
 $1 \leq \text{length (trail (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T))}$

using $\text{cut-trail-wrt-clause-hd-trail-in-or-empty-trail[of mset-ccls } C \text{ } T]$ **by** auto

then show $?thesis$

proof cases

case false **note** $C = \text{this(1)}$ **and** $\text{empty-tr} = \text{this(2)}$

then have $[\text{simp}]: \text{mset-ccls } C = \{\#\}$

by $(\text{simp add: in-CNot-implies-uminus(2) multiset-eqI})$

show $?thesis$

using $\text{empty-tr unfolding cdcl}_W\text{-stgy-invariant-def no-smaller-conflict-def}$
 $\text{cdcl}_W\text{-all-struct-inv-def}$ **by** $(\text{auto simp: add-new-clause-and-update-def})$

next

case not-false **note** $C = \text{this(1)}$ **and** $l = \text{this(2)}$

let $?L = - \text{lit-of (hd (trail (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T)))}$

have $\text{get-all-levels-of-marked (trail (add-new-clause-and-update } C \text{ } T)) =$

$\text{rev [1..<1 + length (get-all-levels-of-marked (trail (add-new-clause-and-update } C \text{ } T))])}$

using $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$ **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$
by blast

moreover

have $\text{backtrack-lvl (cut-trail-wrt-clause (mset-ccls } C) (\text{trail } T) \text{ } T) =$

$\text{length (get-all-levels-of-marked (trail (add-new-clause-and-update } C \text{ } T)))}$

using $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$ **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$
by $(\text{auto simp: add-new-clause-and-update-def})$

moreover
have *no-dup* (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T))
using $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$ **unfolding** *cdcl_W-all-struct-inv-def* *cdcl_W-M-level-inv-def*
by (auto simp:add-new-clause-and-update-def)
then have *atm-of* ?L \notin *atm-of* ' lits-of-l
(tl (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)))
by (cases trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)) auto

ultimately have L: *get-level* (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)) (−?L)
= *length* (*get-all-levels-of-marked* (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)))
using *get-level-get-rev-level-get-all-levels-of-marked*[OF
 $\langle \text{atm-of } ?L \notin \text{atm-of ' lits-of-l (tl (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)))} \rangle$,
of [hd (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T))]]

apply (cases trail (add-init-cls (cls-of-ccls C)
(cut-trail-wrt-clause (mset-ccls C) (trail T) T));
cases hd (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) T)))
using l **by** (auto split: if-split-asm
simp:rev-swap[symmetric] add-new-clause-and-update-def)

have L': *length* (*get-all-levels-of-marked* (trail (cut-trail-wrt-clause (mset-ccls C)
(trail T) T)))
= *backtrack-lvl* (cut-trail-wrt-clause (mset-ccls C) (trail T) T)
using $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$ **unfolding** *cdcl_W-all-struct-inv-def* *cdcl_W-M-level-inv-def*
by (auto simp:add-new-clause-and-update-def)

have [simp]: *no-smaller-confl* (*update-conflicting* (Some C)
(add-init-cls (cls-of-ccls C) (cut-trail-wrt-clause (mset-ccls C) (trail T) T)))
unfolding *no-smaller-confl-def*
proof (*clarify*, *goal-cases*)
case (1 M K i M' D)
then consider
(DC) D = mset-ccls C
| (D-T) D \in # clauses T
by (auto simp: raw-clauses-def split: if-split-asm)
then show False
proof cases
case D-T
have *no-smaller-confl* T
using *inv-s* **unfolding** *cdcl_W-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 $\langle \text{no-smaller-confl } T \rangle$ 1(3) **unfolding** *no-smaller-confl-def* **by** blast
next
case DC **note** $\neg[\text{simp}] = \text{this}$
then have *atm-of* (−?L) \in *atm-of* ' (lits-of-l M)
using 1(3) C *in-CNot-implies-uminus*(2) **by** blast
moreover
have *lit-of* (hd (M' @ Marked K i # [])) = −?L
using l 1(1)[symmetric] *inv*
by (cases trail (add-init-cls (cls-of-ccls C)
(cut-trail-wrt-clause (mset-ccls C) (trail T) T)))
(auto dest!: *arg-cong*[of - # - - hd] simp: *hd-append* *cdcl_W-all-struct-inv-def*
cdcl_W-M-level-inv-def)

```

    from arg-cong[OF this, of atm-of]
    have atm-of  $(-?L) \in \text{atm-of } \langle \text{lits-of-l } (M' @ \text{Marked } K \ i \ \# \ []) \rangle$ 
      by (cases  $(M' @ \text{Marked } K \ i \ \# \ [])$ ) auto
    moreover have no-dup (trail (cut-trail-wrt-clause (mset-ccls C) (trail T) 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 simp: add-new-clause-and-update-def)
    ultimately show False
      unfolding 1(1)[symmetric, simplified] by auto
  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 cdclW-stgy S T and
    inv-s: cdclW-stgy-invariant S and
    inv: cdclW-all-struct-inv S
  shows conflicting T = Some {#}  $\wedge$  unsatisfiable (set-mset (init-clss S))
     $\vee$  conflicting T = None  $\wedge$  trail T  $\models_{\text{asm}}$  init-clss S  $\wedge$  satisfiable (set-mset (init-clss S))
proof -
  have no-step cdclW-stgy T
    using full unfolding full-def by blast
  moreover have cdclW-all-struct-inv T and inv-s: cdclW-stgy-invariant T
    apply (metis rtranclp-cdclW-stgy-rtranclp-cdclW full full-def inv
      rtranclp-cdclW-all-struct-inv-inv)
    by (metis full full-def inv inv-s rtranclp-cdclW-stgy-cdclW-stgy-invariant)
  ultimately have conflicting T = Some {#}  $\wedge$  unsatisfiable (set-mset (init-clss T))
     $\vee$  conflicting T = None  $\wedge$  trail T  $\models_{\text{asm}}$  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-l (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 = init-clss T
    using inv unfolding cdclW-all-struct-inv-def
    by (metis rtranclp-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-confl C T)

```

```

let ?T = (update-conflicting (Some C) (add-init-cls (cls-of-ccls C)
  (cut-trail-wrt-clause (mset-ccls C) (trail S) S)))
have cdclW-all-struct-inv ?T and inv-s-T: cdclW-stgy-invariant ?T
  using add-confli.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-confli.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-confli.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-confli.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-confli C T)
case 1
have cdclW-all-struct-inv (add-init-cls (cls-of-ccls C) S)
  using inv <distinct-mset (mset-ccls C)> unfolding cdclW-all-struct-inv-def no-strange-atm-def
  cdclW-M-level-inv-def distinct-cdclW-state-def cdclW-conflicting-def cdclW-learned-clause-def
  by (auto 9 1 simp: all-decomposition-implies-insert-single raw-clauses-def)

then show ?case
  using add-no-confli(5) unfolding full-def by (auto intro: rtrancpl-cdclW-stgy-cdclW-all-struct-inv)
case 2
have nc:  $\forall M. (\exists K i M'. \text{trail } S = M' @ \text{Marked } K i \# M) \longrightarrow \neg M \models_{as} C \text{Not } (mset-ccls C)$ 
  using < $\neg \text{trail } S \models_{as} C \text{Not } (mset-ccls C)$ >
  by (auto simp: true-annots-true-cls-def-iff-negation-in-model)

have cdclW-stgy-invariant (add-init-cls (cls-of-ccls C) S)
  using s-inv < $\neg \text{trail } S \models_{as} C \text{Not } (mset-ccls C)$ > inv unfolding cdclW-stgy-invariant-def
  no-smaller-confli-def eq-commute[of - trail -] cdclW-M-level-inv-def cdclW-all-struct-inv-def
  by (auto simp: raw-clauses-def nc)
then show ?case
  by (metis <cdclW-all-struct-inv (add-init-cls (cls-of-ccls C) S)> add-no-confli.hyps(5) full-def
    rtrancpl-cdclW-stgy-cdclW-stgy-invariant)
qed

```

```

lemma rtrancpl-incremental-cdclW-inv:
  assumes
    inc: incremental-cdclW** S T and
    inv: cdclW-all-struct-inv S and
    s-inv: cdclW-stgy-invariant S
  shows
    cdclW-all-struct-inv T and
    cdclW-stgy-invariant T
  using inc apply induction
  using inv apply simp
  using s-inv apply simp
  using incremental-cdclW-inv by blast+

```

```

lemma incremental-conclusive-state:
  assumes

```

```

    inc: incremental-cdclW S T and
    inv: cdclW-all-struct-inv S and
    s-inv: cdclW-stgy-invariant S
  shows conflicting T = Some {#} ∧ unsatisfiable (set-mset (init-clss T))
    ∨ conflicting T = None ∧ trail T ⊨asm init-clss T ∧ satisfiable (set-mset (init-clss T))
  using inc
proof induction
  print-cases
  case (add-confl C T) note tr = this(1) and dist = this(2) and conf = this(3) and C = this(4) and
    full = this(5)

  have full cdclW-stgy T T
    using full unfolding full-def by auto
  then show ?case
    using full C conf dist tr
    by (metis full-cdclW-stgy-inv-normal-form incremental-cdclW.simps incremental-cdclW-inv(1)
        incremental-cdclW-inv(2) inv s-inv)
next
  case (add-no-confl C T) note tr = this(1) and dist = this(2) and conf = this(3) and C = this(4)
    and full = this(5)

  have full cdclW-stgy T T
    using full unfolding full-def by auto
  then show ?case
    by (meson C conf dist full full-cdclW-stgy-inv-normal-form incremental-cdclW.add-no-confl
        incremental-cdclW-inv(1) incremental-cdclW-inv(2) inv s-inv tr)
qed

lemma tranclp-incremental-correct:
  assumes
    inc: incremental-cdclW++ S T and
    inv: cdclW-all-struct-inv S and
    s-inv: cdclW-stgy-invariant S
  shows conflicting T = Some {#} ∧ unsatisfiable (set-mset (init-clss T))
    ∨ conflicting T = None ∧ trail T ⊨asm init-clss T ∧ satisfiable (set-mset (init-clss T))
  using inc apply induction
    using assms incremental-conclusive-state apply blast
  by (meson incremental-conclusive-state inv rtranclp-incremental-cdclW-inv s-inv
      tranclp-into-rtranclp)

end

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin
sledgehammer-params[verbose]
no-notation Herbrand-Interpretation.true-cls (infix ⊨ 50)
notation Herbrand-Interpretation.true-cls (infix ⊨h 50)

no-notation Herbrand-Interpretation.true-clss (infix ⊨s 50)
notation Herbrand-Interpretation.true-clss (infix ⊨hs 50)

lemma herbrand-interp-iff-partial-interp-cls:
  S ⊨h C ⟷ {Pos P | P. P ∈ S} ∪ {Neg P | P. P ∉ S} ⊨ C

```

unfolding *Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def*
by *auto*

lemma *herbrand-consistent-interp:*
consistent-interp ($\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$)
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* =
 $\{A. C \in N \wedge C \neq \{\#\} \wedge Max\ (set_mset\ C) = Pos\ A \wedge count\ C\ (Pos\ A) \leq 1$
 $\wedge \neg (\bigcup D \in \{D. D \# \subset \# C\}. production\ D) \models_h C \wedge S\ C = \{\#\}\}$
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* = $(\bigcup D \in \{D. D \# \subset \# C\}. production\ D)$

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* $A \equiv (\text{production } A \neq \{\})$

abbreviation *produces* $:: 'a \text{ clause} \Rightarrow 'a \Rightarrow \text{bool}$ **where**

produces $C A \equiv \text{production } C = \{A\}$

lemma *producesD*:

produces $C A \Longrightarrow C \in N \wedge C \neq \{\#\} \wedge \text{Pos } A = \text{Max} (\text{set-mset } C) \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$
interp $C \models_h C \wedge S C = \{\#\}$

unfolding *production-unfold* **by** *auto*

lemma *produces* $C A \Longrightarrow \text{Pos } A \in \# C$

by (*simp add: Max-in-lits producesD*)

lemma *interp'-def-in-set*:

interp $C = (\bigcup D \in \{D \in N. D \# \subseteq \# C\}. \text{production } D)$

unfolding *interp-def* **apply** *auto*

unfolding *production-unfold* **apply** *auto*

done

lemma *production-iff-produces*:

produces $D A \longleftrightarrow A \in \text{production } D$

unfolding *production-unfold* **by** *auto*

definition *Interp* $:: 'a \text{ clause} \Rightarrow 'a \text{ interp}$ **where**

Interp $C = \text{interp } C \cup \text{production } C$

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* $:: 'a \text{ interp}$ **where**

INTERP $= (\bigcup D \in N. \text{production } D)$

lemma *interp-subseteq-Interp*[*simp*]: *interp* $C \subseteq \text{Interp } C$

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* $C \Longrightarrow C \neq \{\#\}$

unfolding *production-unfold* **by** *auto*

lemma *productive-imp-produces-Max-literal*: *productive* $C \Longrightarrow \text{produces } C (\text{atm-of } (\text{Max} (\text{set-mset } C)))$

unfolding *production-unfold* **by** (*auto simp del: atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom*: *productive* $C \Longrightarrow \text{produces } C (\text{Max} (\text{atms-of } C))$

unfolding *atms-of-def* *Max-atm-of-set-mset-commute*[*OF productive-not-empty*]

by (rule productive-imp-produces-Max-literal)

lemma produces-imp-Max-literal: produces C $A \implies A = \text{atm-of } (\text{Max } (\text{set-mset } C))$
 by (metis Max-singleton insert-not-empty productive-imp-produces-Max-literal)

lemma produces-imp-Max-atom: produces C $A \implies A = \text{Max } (\text{atms-of } C)$
 by (metis Max-singleton insert-not-empty productive-imp-produces-Max-atom)

lemma produces-imp-Pos-in-lits: produces C $A \implies \text{Pos } A \in\# C$
 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 $A \implies B \in \text{atms-of } C \implies B \leq A$
 by (metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom singleton-inject)

lemma produces-imp-neg-notin-lits: produces C $A \implies \neg \text{Neg } A \in\# C$
 by (rule pos-Max-imp-neg-notin) (auto dest: producesD)

lemma less-eq-imp-interp-subseteq-interp: $C \# \subseteq\# D \implies \text{interp } C \subseteq \text{interp } D$
 unfolding interp-def by auto (metis multiset-order.order.strict-trans2)

lemma less-eq-imp-interp-subseteq-Interp: $C \# \subseteq\# D \implies \text{interp } C \subseteq \text{Interp } D$
 unfolding Interp-def using less-eq-imp-interp-subseteq-interp by blast

lemma less-imp-production-subseteq-interp: $C \# \subset\# D \implies \text{production } C \subseteq \text{interp } D$
 unfolding interp-def by fast

lemma less-eq-imp-production-subseteq-Interp: $C \# \subseteq\# D \implies \text{production } C \subseteq \text{Interp } D$
 unfolding Interp-def using less-imp-production-subseteq-interp
 by (metis multiset-order.le-imp-less-or-eq le-supI1 sup-ge2)

lemma less-imp-Interp-subseteq-interp: $C \# \subset\# D \implies \text{Interp } C \subseteq \text{interp } D$
 unfolding Interp-def
 by (auto simp: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp)

lemma less-eq-imp-Interp-subseteq-Interp: $C \# \subseteq\# D \implies \text{Interp } C \subseteq \text{Interp } D$
 using less-imp-Interp-subseteq-interp
 unfolding Interp-def by (metis multiset-order.le-imp-less-or-eq le-supI2 subset-refl sup-commute)

lemma false-Interp-to-true-interp-imp-less-multiset: $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \# \subset\# D$
 using less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less by blast

lemma false-interp-to-true-interp-imp-less-multiset: $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \# \subset\# D$
 using less-eq-imp-interp-subseteq-interp multiset-linorder.not-less by blast

lemma false-Interp-to-true-Interp-imp-less-multiset: $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \# \subset\# D$
 using less-eq-imp-Interp-subseteq-Interp multiset-linorder.not-less by blast

lemma false-interp-to-true-Interp-imp-le-multiset: $A \notin \text{interp } C \implies A \in \text{Interp } D \implies C \# \subseteq\# D$
 using less-imp-Interp-subseteq-interp multiset-linorder.not-less by blast

lemma interp-subseteq-INTERP: $\text{interp } C \subseteq \text{INTERP}$

unfolding *interp-def* *INTERP-def* **by** (*auto simp: production-unfold*)

lemma *production-subseteq-INTERP*: *production* $C \subseteq \text{INTERP}$
unfolding *INTERP-def* **using** *production-unfold* **by** *blast*

lemma *Interp-subseteq-INTERP*: *Interp* $C \subseteq \text{INTERP}$
unfolding *Interp-def* **by** (*auto intro!: interp-subseteq-INTERP production-subseteq-INTERP*)

This lemma corresponds to theorem 2.7.6 page 66 of CW.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: $\text{Neg } A \in\# C$ **and** *d*: *produces* $D A$
shows $A \in \text{interp } C$
proof –
from *d* **have** $\text{Max } (\text{set-mset } D) = \text{Pos } A$
using *production-unfold* **by** *blast*
hence $D \# \subset \# \{\# \text{Neg } A \#\}$
by (*auto intro: Max-pos-neg-less-multiset*)
moreover **have** $\{\# \text{Neg } A \#\} \# \subseteq \# C$
by (*rule less-eq-imp-le-multiset*) (*rule mset-le-single[OF a-in-c]*)
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*: $\text{Neg } A \in\# C \implies A \notin \text{Interp } D \implies C \# \subseteq \# D \implies \neg \text{produces } D'' A$
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* $\exists A. \text{Pos } A \in\# C \wedge A \in \text{Interp } D$)
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-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** $a\text{-in-}c$: $Neg\ A \in\# C$ **and** $A \notin Interp\ D$
using $c\text{-at-}d$ **unfolding** $true\text{-cls-def}$ **by** $blast$
hence $\bigwedge D''. \neg\ produces\ D''\ A$
using $c\text{-le-}d$ $neg\text{-notin-}Interp\text{-not-produce}$ **by** $simp$
thus $?thesis$
using $a\text{-in-}c$ $subs\ not\text{-produces-}imp\text{-notin-production}$ **by** $auto$
qed

lemma $true\text{-Interp-imp-interp}$: $C \# \subseteq\# D \implies D \# \subset\# D' \implies Interp\ D \models_h C \implies interp\ D' \models_h C$
using $interp\text{-def}$ $true\text{-Interp-imp-general}$ **by** $simp$

lemma $true\text{-Interp-imp-Interp}$: $C \# \subseteq\# D \implies D \# \subset\# D' \implies Interp\ D \models_h C \implies Interp\ D' \models_h C$
using $Interp\text{-as-UNION}$ $interp\text{-subsetq-Interp}$ $true\text{-Interp-imp-general}$ **by** $simp$

lemma $true\text{-Interp-imp-INTERP}$: $C \# \subseteq\# D \implies Interp\ D \models_h C \implies INTERP \models_h C$
using $INTERP\text{-def}$ $interp\text{-subsetq-INTERP}$
 $true\text{-Interp-imp-general}[OF\ -\ less\text{-multiset-right-total}]$
by $simp$

lemma $true\text{-interp-imp-general}$:

assumes

$c\text{-le-}d$: $C \# \subseteq\# D$ **and**

$d\text{-lt-}d'$: $D \# \subset\# D'$ **and**

$c\text{-at-}d$: $interp\ D \models_h C$ **and**

$subs$: $interp\ D' \subseteq (\bigcup C \in CC.\ production\ C)$

shows $(\bigcup C \in CC.\ production\ C) \models_h C$

proof ($cases\ \exists A.\ Pos\ A \in\# C \wedge A \in interp\ D$)

case $True$

then obtain A **where** $a\text{-in-}c$: $Pos\ A \in\# C$ **and** $a\text{-at-}d$: $A \in interp\ D$

by $blast$

from $a\text{-at-}d$ **have** $A \in interp\ D'$

using $d\text{-lt-}d'$ $less\text{-eq-imp-interp-subsetq-interp}[OF\ multiset\text{-order.less-imp-le}]$ **by** $blast$

thus $?thesis$

using $subs\ a\text{-in-}c$ **by** ($blast\ dest$: $contra\text{-subset}D$)

next

case $False$

then obtain A **where** $a\text{-in-}c$: $Neg\ A \in\# C$ **and** $A \notin interp\ D$

using $c\text{-at-}d$ **unfolding** $true\text{-cls-def}$ **by** $blast$

hence $\bigwedge D''. \neg\ produces\ D''\ A$

using $c\text{-le-}d$ **by** ($auto\ dest$: $produces\text{-imp-in-}interp\ less\text{-eq-imp-interp-subsetq-interp$)

thus $?thesis$

using $a\text{-in-}c$ $subs\ not\text{-produces-}imp\text{-notin-production}$ **by** $auto$

qed

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

lemma $true\text{-interp-imp-interp}$: $C \# \subseteq\# D \implies D \# \subset\# D' \implies interp\ D \models_h C \implies interp\ D' \models_h C$
using $interp\text{-def}$ $true\text{-interp-imp-general}$ **by** $simp$

lemma $true\text{-interp-imp-Interp}$: $C \# \subseteq\# D \implies D \# \subset\# D' \implies interp\ D \models_h C \implies Interp\ D' \models_h C$
using $Interp\text{-as-UNION}$ $interp\text{-subsetq-Interp}[of\ D']$ $true\text{-interp-imp-general}$ **by** $simp$

lemma $true\text{-interp-imp-INTERP}$: $C \# \subseteq\# D \implies interp\ D \models_h C \implies INTERP \models_h C$
using $INTERP\text{-def}$ $interp\text{-subsetq-INTERP}$
 $true\text{-interp-imp-general}[OF\ -\ less\text{-multiset-right-total}]$
by $simp$

lemma *productive-imp-false-interp*: $productive\ C \implies \neg\ interp\ C \models_h C$
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
 $(P)\ count\ C\ (Pos\ P) \geq 2$
 $| (Q)\ Q\ where\ Q > Pos\ P\ and\ Q \in \# C$
using $HOL.spec[OF\ HOL.conjunct2[OF\ D'],\ of\ Pos\ P]$ **by** *(auto split: if-split-asm)*
thus $?thesis$
proof *cases*
case Q
have $Q \in set-mset\ C$
using $Q(2)$ **by** *(auto split: if-split-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
 $(P)\ Neg\ P \in \# D$
 $| (Q)\ Q\ where\ Q > Neg\ P\ and\ count\ D\ Q > count\ (C + \{\#Neg\ P\# \})\ Q$
using $HOL.spec[OF\ HOL.conjunct2[OF\ D'],\ of\ Neg\ P]\ count-greater-zero-iff$ **by** *fastforce*
thus $?thesis$
proof *cases*
case Q
have $Q \in set-mset\ D$
using $Q(2)\ gr-implies-not0$ **by** *fastforce*
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 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 $MMax\ M \equiv Max\ (set-mset\ M)$

22.1 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 $?I = \{Pos\ P \mid P. P \in A\} \cup \{Neg\ P \mid P. P \notin A\}$

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: *abstract-red C N* **and**

c-lt-d: $C \subseteq\# D$

shows *abstract-red D N*

proof –

have $\{D \in N. D \# \subset\# C\} \subseteq \{D' \in N. D' \# \subset\# D\}$

using *c-lt-d less-eq-imp-le-multiset* **by** *fastforce*

thus *?thesis*

using *abstr unfolding abstract-red-def clss-lt-def*

by (*metis (no-types, lifting) c-lt-d subset-mset.diff-add true-clss-cls-mono-r'*
true-clss-cls-subset)

qed

lemma *true-clss-cls-extended*:

assumes

$A \models_p B$ **and**

tot: *total-over-m I (A)* **and**

cons: *consistent-interp I* **and**

I-A: $I \models_s A$

shows $I \models B$

proof –

let $?I = I \cup \{Pos P \mid P. P \in atms-of B \wedge P \notin atms-of-s I\}$

have *consistent-interp ?I*

using *cons unfolding consistent-interp-def atms-of-s-def atms-of-def*

apply (*auto 1 5 simp add: image-iff*)

by (*metis atm-of-uminus literal.sel(1)*)

moreover have *total-over-m ?I (A \cup {B})*

proof –

obtain *aa* :: '*a* set \Rightarrow '*a* literal set \Rightarrow '*a* **where**

f2: $\forall x0 x1. (\exists v2. v2 \in x0 \wedge Pos v2 \notin x1 \wedge Neg v2 \notin x1)$

$\longleftrightarrow (aa x0 x1 \in x0 \wedge Pos (aa x0 x1) \notin x1 \wedge Neg (aa x0 x1) \notin x1)$

by *moura*

have $\forall a. a \notin atms-of-ms A \vee Pos a \in I \vee Neg a \in I$

using *tot* **by** (*simp add: total-over-m-def total-over-set-def*)

hence *aa* (*atms-of-ms A \cup atms-of-ms {B}*) (*I \cup {Pos a | a. a \in atms-of B \wedge a \notin atms-of-s I}*)
 $\notin atms-of-ms A \cup atms-of-ms \{B\} \vee Pos (aa (atms-of-ms A \cup atms-of-ms \{B\})$

(*I \cup {Pos a | a. a \in atms-of B \wedge a \notin atms-of-s I}*)) $\in I$

$\cup \{Pos a \mid a. a \in atms-of B \wedge a \notin atms-of-s I\}$

$\vee Neg (aa (atms-of-ms A \cup atms-of-ms \{B\})$

(*I \cup {Pos a | a. a \in atms-of B \wedge a \notin atms-of-s I}*)) $\in I$

$\cup \{Pos a \mid a. a \in atms-of B \wedge a \notin atms-of-s I\}$

by *auto*

hence *total-over-set (I \cup {Pos a | a. a \in atms-of B \wedge a \notin atms-of-s I}) (atms-of-ms A \cup 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-clss-cls-def* **by** *auto*

```

thus ?thesis
oops
lemma
  assumes
    CP:  $\neg \text{clss-lt } N (\{ \#C\# \} + \{ \#E\# \}) \models_p \{ \#C\# \} + \{ \#Neg P\# \}$  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
  saturated  $N \longleftrightarrow (\forall A \ B \ C. A \in N \longrightarrow B \in N \longrightarrow \neg \text{redundant } A \ N \longrightarrow \neg \text{redundant } B \ N$ 
 $\longrightarrow \text{superposition-rules } A \ B \ C \longrightarrow \text{redundant } C \ N \vee C \in N)$ 

lemma
  assumes
    saturated: saturated  $N$  and
    finite: finite  $N$  and
    empty:  $\{ \# \} \notin N$ 
  shows  $\text{INTERP } N \models_{hs} N$ 
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:  $\bigwedge E. E \in N \implies E \neq D \implies \neg ?N_{\mathcal{I}} \models_h E \implies D \leq E$ 
  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 ∈# 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: ¬redundant D N
proof (rule ccontr)
  assume red[simplified]: ~redundant D N
  have ∀ E < D. E ∈ N ⟶ ?NI ⊨h E
    using cls-not-D not-le by fastforce
  hence ?NI ⊨hs clss-lt N D
    unfolding clss-lt-def true-clss-def Ball-def by blast
  thus False
    using red not-d-interp unfolding abstract-red-def redundant-iff-abstract
    using herbrand-true-clss-true-clss-cls-herbrand-true-clss by fast
qed

consider
  (L) P where L = 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 L D] 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 ~ ?thesis
    hence count: count D L = 1
    unfolding D by (auto simp: not-in-iff)
    have ¬?NI ⊨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 ⟨D ∈ N⟩ count L
    true-interp-imp-INTERP unfolding production-iff-produces unfolding production-unfold
    by (auto simp add: max not-empty)
    hence INTERP N ⊨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 have Pos P ∈# C
  by (simp add: P D)
then obtain C' where C':D = C' + {# Pos P #} + {# Pos P #}
  unfolding D by (metis (full-types) P insert-DiffM2)
have sup: superposition-rules D D (D - {# L #})
  unfolding C' L by (auto simp add: superposition-rules.simps)
have C' + {# Pos P #} #⊂# C' + {# Pos P #} + {# Pos P #}
  by auto
moreover have ¬?NI ⊨h (D - {# L #})

```

```

    using not-d-interp unfolding C' L by auto
ultimately have C' + {#Pos P#} ∉ 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#} #⊂# 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 ⟨D ∈ N⟩⟨C' + {#Pos P#} ∉ N⟩ unfolding saturated-def C' L c'-p-p
  by blast
moreover have C' + {#Pos P#} ⊆# 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 ∈ ?NI
  using not-d-interp unfolding D true-cls-def L by (auto split: if-split-asm)
then obtain E where
  DPN: E + {#Pos P#} ∈ 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 ∪ {E+C})
  using DPN ⟨D ∈ N⟩ unfolding D L by (auto simp add: superposition.simps)
have
  PMax: Pos P = MMax (E + {#Pos P#}) and
  count (E + {#Pos P#}) (Pos P) ≤ 1 and
  S (E + {#Pos P#}) = {#} and
  ¬interp N (E + {#Pos P#}) ⊢h E + {#Pos P#}
  using prod unfolding production-unfold by auto
have Neg P ∉# E
  using prod produces-imp-neg-notin-lits by force
hence ∧y. y ∈# (E + {#Pos P#})
  ⇒ count (E + {#Pos P#}) (Neg P) < count (C + {#Neg P#}) (Neg P)
  using count-greater-zero-iff by fastforce
moreover have ∧y. y ∈# (E + {#Pos P#}) ⇒ y < Neg P
  using PMax by (metis DPN Max-less-iff empty finite-set-mset pos-less-neg
      set-mset-eq-empty-iff)
moreover have E + {#Pos P#} ≠ C + {#Neg P#}
  using prod produces-imp-neg-notin-lits by force
ultimately have E + {#Pos P#} #⊂# C + {#Neg P#}
  unfolding less-multisetHO by (metis count-greater-zero-iff less-iff-Suc-add zero-less-Suc)
have ce-lt-d: C + E #⊂# D
  unfolding D L by (simp add: ⟨∧y. y ∈# E + {#Pos P#} ⇒ y < Neg P⟩ ex-gt-imp-less-multiset)
have ?NI ⊢h E + {#Pos P#}
  using ⟨P ∈ ?NI⟩ by blast
have ?NI ⊢h C+E ∨ C+E ∉ N
  using ce-lt-d cls-not-D unfolding D-def by fastforce
have Pos P ∉# C+E
  using D ⟨P ∈ ground-resolution-with-selection.INTERP S N⟩
      ⟨count (E + {#Pos P#}) (Pos P) ≤ 1⟩ multi-member-skip not-d-interp

```

```

    by (auto simp: not-in-iff)
  hence  $\bigwedge y. y \in \# C + E$ 
     $\implies \text{count } (C + E) (\text{Pos } P) < \text{count } (E + \{\# \text{Pos } P\# \}) (\text{Pos } P)$ 
    using set-mset-def by fastforce

  have  $\neg \text{redundant } (C + E) N$ 
  proof (rule ccontr)
    assume  $\text{red}'[\text{simplified}]: \neg ?thesis$ 
    have  $\text{abs}: \text{clss-lt } N (C + E) \models_p C + E$ 
      using redundant-iff-abstract  $\text{red}'$  unfolding abstract-red-def by auto
    have  $\text{clss-lt } N (C + E) \models_p E + \{\# \text{Pos } P\# \} \vee \text{clss-lt } N (C + E) \models_p C + \{\# \text{Neg } P\# \}$ 
    proof clarify
      assume  $CP: \neg \text{clss-lt } N (C + E) \models_p C + \{\# \text{Neg } P\# \}$ 
      { fix I
        assume
          total-over-m I ( $\text{clss-lt } N (C + E) \cup \{E + \{\# \text{Pos } P\# \}\}$ ) and
          consistent-interp I and
           $I \models_s \text{clss-lt } N (C + E)$ 
        hence  $I \models C + E$ 
          using abs sorry
        moreover have  $\neg I \models C + \{\# \text{Neg } P\# \}$ 
          using CP unfolding true-clss-cls-def
          sorry
        ultimately have  $I \models E + \{\# \text{Pos } P\# \}$  by auto
      }
      then show  $\text{clss-lt } N (C + E) \models_p E + \{\# \text{Pos } P\# \}$ 
        unfolding true-clss-cls-def by auto
    qed
    moreover have  $\text{clss-lt } N (C + E) \subseteq \text{clss-lt } N (C + \{\# \text{Neg } P\# \})$ 
      using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
    ultimately have  $\text{redundant } (C + \{\# \text{Neg } P\# \}) N \vee \text{clss-lt } N (C + E) \models_p E + \{\# \text{Pos } P\# \}$ 
      unfolding redundant-iff-abstract abstract-red-def using true-clss-cls-subset by blast
    show False sorry
  qed
  moreover have  $\neg \text{redundant } (E + \{\# \text{Pos } P\# \}) N$ 
    sorry
  ultimately have  $CEN: C + E \in N$ 
    using  $\langle D \in N \rangle \langle E + \{\# \text{Pos } P\# \} \in N \rangle$  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  $\text{interp}: \neg \text{INTERP } N \models_h C + E$ 
    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-cls-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 :: wellorder clause and N :: 'a :: wellorder clauses
  assumes redundant A N
  shows abstract-red A N
  using assms
proof (induction rule: redundant.induct)
  case (subsumption A B N)
  thus ?case
    using subsumed-is-redundant[of A N B] unfolding abstract-red-def 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 :: nat clause  $\Rightarrow$  nat clause
begin

end

end
theory Weidenbach-Book
imports
  Prop-Normalisation

  Prop-Resolution

  Prop-Superposition

  CDCL-NOT DPLL-NOT DPLL-W-Implementation CDCL-W-Implementation CDCL-W-Incremental
  CDCL-WNOT

begin

end

```